Eelco Visser Commemorative Symposium

EVCS 2023, April 5, 2023, Delft, The Netherlands

Edited by

Ralf Lämmel
Peter D. Mosses
Friedrich Steimann

OASICS

Editors

Ralf Lämmel
University of Koblenz, Germany
laemmel@uni-koblenz.de

Peter D. Mosses
Delft University of Technology, The Netherlands
P.D.Mosses@tudelft.nl

Friedrich Steimann
Fernuniversität in Hagen, Germany
steimann@acm.org

ACM Classification 2012
Software and its engineering; Theory of computation; Information systems; Mathematics of computing; Social and professional topics; Applied computing → Education; General and reference → Biographies

ISBN 978-3-95977-267-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-267-9.

Publication date
March, 2023

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.EVCS.2023.0

ISBN 978-3-95977-267-9 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Keynote Talk

Regular Papers

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

Eelco Visser (12 October 1966 – 5 April 2022) was Antoni van Leeuwenhoek Professor of Computer Science and Chair of the Programming Languages Group in the Department of Software Technology at TU Delft. He obtained his Master's degree and PhD in Computer Science from the University of Amsterdam, supervised by Paul Klint. He subsequently held positions at Oregon Graduate Institute, Utrecht University, and TU Delft. He was a founding member of IFIP Working Groups 2.11 (Program Generation) and 2.16 (Programming Language Design).

Eelco was highly influential in the software language engineering and programming language design communities. His many scientific contributions on meta- and domain-specific languages have been of high importance in both the scientific and industrial communities. His work on the cutting-edge language workbench *Spoofax* started with a ground-breaking publication in 2010, for which he received a *Most Influential Paper* award at OOPSLA 2020. As a strong advocate of tool-supported programming education, he led the development of *WebLab*, a learning management system that is in use for a range of programming languages and courses at TU Delft. He also led the design, implementation and use of conf.researchr.org, a content management system for scientific events used for hundreds of international events since 2014.

On 5 April 2023, the first anniversary of Eelco's untimely passing, many of his collaborators, peers, and friends are gathering at TU Delft to hold the Eelco Visser Commemorative Symposium (EVCS). The proceedings of the EVCS preserve the spirit of this event: cherishing the memory of Eelco as a researcher, as a developer, and as a leader. The reader will easily recognise all three sides of Eelco when studying the individual contributions captured in these proceedings, and will grasp the immensity of his loss and what it means to the affected communities.

Conceiving, planning, organising, and carrying out the EVCS, as well as writing, reviewing and publishing its 32 contributions, was the joint effort of a great many. The communities of SLE, GPCE, OOPSLA (SPLASH), IFIP WG 2.11 and 2.16, as well as CWI and TU Delft, supported the EVCS, and delegated representatives to its Programme and Organising Committee. The reviewing of the submissions that followed the call for papers involved 35 external reviewers, whose diligence and constructiveness helped shape the final contributions. We are especially grateful to the Department of Software Technology, TU Delft, for funding the EVCS, and to the local organisation committee for logistic and administrative support. The Dagstuhl Publishing team made producing these proceedings as a volume of the Open Access Series in Informatics an exceptionally smooth experience.

Helping to make the EVCS happen was very dear to us. We hope that those attending the symposium, and reading its proceedings, will be rewarded with many recollections of Eelco and his academic activities, and with inspiration for their own future work – work which, sadly, Eelco himself is denied. His memory and influence will stay with us.

Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann

Editors

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Programme and Organising Committee

Jonathan Aldrich
Carnegie Mellon University

Benoit Combemale
University of Rennes, Inria, and IRISA

Paul Klint
CWI and University of Amsterdam

Ralf Lämmel
University of Koblenz (Program Co-chair)

Peter Mosses
TU Delft and Swansea University
(Organizing Chair)

Friedrich Steimann
Fernuniversität in Hagen (Program Co-chair)

Tijs van der Storm
CWI and University of Groningen

Eric Van Wyk
University of Minnesota

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Additional Reviewers

Casper Bach Poulsen

Erwan Bousse

Thomas Degueule

Luis Eduardo de Souza Amorim

Bernd Fischer

Matthew Flatt

Daco Harkes

Görel Hedin

Robert Hirschfeld

John Hughes

Adrian Johnstone

Yukiyoshi Kameyama

Lennart Kats

Oleg Kiselyov

Julia Lawall

Gurvan Le Guernic

Daan Leijen

José Nuno Oliveira

François Pottier

Max Schäfer

Ulrik Pagh Schultz Lundquist

Philipp Schuster

Neil Sculthorpe

Manuel Serrano

Tony Sloane

Yannis Smaragdakis

Jeff Smits

Emma Söderberg

Volker Stolz

Éric Tanter

Andrew Tolmach

Hendrik van Antwerpen

L. Thomas van Binsbergen

Jurgen Vinju

Vadim Zaytsev

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Local Organisation Committee

Arie van Deursen

Sophie den Hartog

Peter Mosses

Kim Roos

Shelly Dawn Stok

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Guillaume Allais (9)
University of St Andrews, UK

Craig Anslow (12)
Victoria University of Wellington, New Zealand

Casper Bach Poulsen (2)
Delft University of Technology, The Netherlands

Jason Balaci (7)
Department of Computing and Software,
McMaster University, Hamilton, Canada

Michael Ballantyne (3)
PLT at Northeastern University,
Boston, MA, USA

Mikhail Barash (25)
University of Bergen, Norway

Andrew P. Black (4)
Portland State University, OR, USA

Hamza Boucherit (5)
Oracle Labs, Casablanca, Morocco

Houda Boukham (5)
Mohammed V University in Rabat, Ecole
Mohammadia d'Ingénieurs, Morocco;
Oracle Labs, Casablanca, Morocco

Edwin Brady (9)
University of St Andrews, UK

Jonathan Brouwer (6)
Delft University of Technology, The Netherlands

Kim B. Bruce (4)
Pomona College, Claremont, CA, USA

Jacques Carette (7, 19)
Department of Computing and Software,
McMaster University, Hamilton, Canada

Hassan Chafi (5)
Oracle Labs, Zürich, Switzerland

Dalila Chiadmi (5)
Mohammed V University in Rabat, Ecole
Mohammadia d'Ingénieurs, Morocco

Jesper Cockx (6)
Delft University of Technology, The Netherlands

Douglas A. Creager (8)
GitHub, Southborough, MA, USA

Max M. de Krieger (12, 13)
Delft University of Technology, The Netherlands

Jan de Muijnck-Hughes (9)
University of Glasgow, UK

Arnaud Delamare (5)
Oracle Labs, Zürich, Switzerland

Martijn Dwars (5)
Oracle Labs, Zürich, Switzerland

Sebastian Erdweg (10)
Johannes Gutenberg-Universität Mainz,
Germany

Matthias Felleisen (3)
PLT at Northeastern University,
Boston, MA, USA

Michael Greenberg (11)
Stevens Institute of Technology,
Hoboken, NJ, USA

Danny M. Groenewegen (12, 13)
Delft University of Technology, The Netherlands

Toine Hartman (5)
Oracle Labs, Utrecht, The Netherlands

Sungpack Hong (5)
Oracle Labs, Redwood Shores, CA, USA

Dániel Horpácsi (26)
Faculty of Informatics, Eötvös Loránd
University, Budapest, Hungary

Adrian Johnstone (23)
Department of Computer Science,
Royal Holloway, University of London, UK

Jaakko Järvi (25)
University of Turku, Finland

Karl Trygve Kalleberg (14)
KolibriFX AS, Oslo, Norway

Paul Klint (15)
Centrum Wiskunde & Informatica (CWI),
Amsterdam, The Netherlands

James Koppel (16)
MIT, Cambridge, MA, US

Julia Lawall (18)
Inria, Paris, France

Christian Lengauer (19)
Universität Passau, Germany

Markus Lepper (20)
semantics gGmbH Berlin, Germany

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ralf Lämmel (17)
Universität Koblenz, Germany

Peter D. Mosses (21)
Delft University of Technology, The Netherlands;
Swansea University, UK

James Noble (4)
Creative Research & Programing,
Wellington, NZ

Daniel A. A. Pelsmaeker (12, 13)
Delft University of Technology, The Netherlands

Jaro S. Reinders (22)
Delft University of Technology, The Netherlands

Elizabeth Scott (23)
Department of Computer Science, Royal
Holloway, University of London, UK

Spencer W. Smith (7)
Department of Computing and Software,
McMaster University, Hamilton, Canada

Friedrich Steimann (24)
Fernuniversität in Hagen, Germany

Knut Anders Stokke (25)
University of Bergen, Norway

Simon Thompson (26)
School of Computing, University of Kent,
Canterbury, UK;
Faculty of Informatics, Eötvös Loránd
University, Budapest, Hungary

Andrew Tolmach (27)
Portland State University, OR, USA

Baltasar Trancón y Widemann (20)
Nordakademie Elmshorn, Germany;
semantics gGmbH Berlin, Germany

Hendrik van Antwerpen (8, 32)
GitHub, Amsterdam, The Netherlands

Elmer van Chastelet (12, 13)
Delft University of Technology, The Netherlands

Tijs van der Storm (28, 29)
Centrum Wiskunde & Informatica (CWI),
Amsterdam, The Netherlands;
University of Groningen, The Netherlands

Arie van Deursen (1)
Delft University of Technology, The Netherlands

Oskar van Rest (5)
Oracle, Redwood Shores, CA, USA

Eric Van Wyk (30)
Department of Computer Science and
Engineering, University of Minnesota,
Minneapolis, MN, USA

Jurgen J. Vinju (31)
NWO-I Centrum Wiskunde & Informatica
(CWI), Amsterdam, The Netherlands;
TU Eindhoven, The Netherlands

Guido Wachsmuth (5)
Oracle Labs, Zürich, Switzerland

Aron Zwaan (6, 32)
Delft University of Technology, The Netherlands

Getting Things Done: The Eelco Way

Arie van Deursen
Delft University of Technology, The Netherlands

Abstract

Eelco Visser (1966–2022) was a leading member of the department of Software Technology (ST) of the faculty of Electrical Engineering Mathematics, and Computer Science (EEMCS) of Delft University of Technology. He had a profound influence on the educational programs in computer science at TU Delft, built a highly successful Programming Languages Group from the ground up, and used his research results to develop widely used tools and services that have been used by thousands of students and researchers for more than a decade. He realized all these successes not just alone, but in close collaboration with a range of people, who he convinced to follow his lead. In this short reflection, I look back at his achievements, and at the way in which he worked with others to bring ambitious ideas to successful reality.

2012 ACM Subject Classification Social and professional topics

Keywords and phrases Leadership

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.1

1 Background

A common challenge in empirical software engineering is how to establish the value of a proposed new tool, or, fitting in the context of my dearly missed colleague Eelco Visser[1] in whose memory I write this essay, a domain-specific language. New methods and tools to help developers are proposed continuously, but how can one assess that such tools actually help as intended? Among various (quasi-)experimental approaches to assess this, one is the *removed treatment group design* [11], in which measurements are conducted not only before and while using a tool, but also after taking it away again. If removing the tool makes the developers complain, we can conclude that the tool was of value, and we can make the developers happy by giving the tool back to them.

Since April 2022, I have frequently felt like I was in the middle of such an experiment. Someone, for some reason, had decided they wanted to show how valuable Eelco was for the TU Delft, so they took him away from us. “Can we please stop the experiment and get Eelco back?” is what I think almost every day. Unfortunately, that is not an option.

In this essay, I look back at Eelco’s time in Delft, his extra-ordinary achievements for the organization, and, the personal traits that enabled him and his co-workers to realize his successes.

2 Eelco at TU Delft

In 2006, Eelco Visser joined the TU Delft Software Engineering Research Group as an associate professor. I knew Eelco from our earlier collaborations [1, 3] in Amsterdam, and I was very happy he had accepted our offer to join us in Delft. Within the software engineering section he built up more and more programming languages activities, in such areas as parsing [2, 12], package management [6, 7], model-driven software evolution [5], web programming [13, 8], and language work benches [9, 14]. In 2013, he secured a highly

[1] `https://avandeursen.com/2022/04/15/eelco-visser-1966-2022-en/`

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 1; pp. 1:1–1:4
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

prestigious NWO Vici grant that enabled him to attract and employ multiple PhD students and postdocs. In the same year he was also promoted to the rank of full professor.[2] By 2015, Eelco's programming languages activities had grown so much that it was time to spin off a new section, the *Programming Languages Group*. This group officially started on January 1st, 2016, and, under the leadership of Eelco, grew to a total of five (including Eelco) faculty members and numerous postdocs, support staff, and bachelor, master, and PhD students.

January 1st, 2016, was also the day that I started in a new role, as head of the department of Software Technology. At that time, the department consisted of six sections, including the two aforementioned programming languages and software engineering sections. In his role as section leader, Eelco was also part of the *Management Team* of the department. Within this team, we discussed and decided about such issues as opening up our bachelor program to non-Dutch speaking students, handling the enormous increase in student intake in 2018, navigating students and employees safely through COVID-19, the increasingly prominent role of artificial intelligence and its impact on computer science education, and hiring and promotion decisions, effectively doubling the department in size over the years.

3 Lessons Learned from Eelco

Eelco played a leading role in Delft, both through the Programming Languages Group that he founded, and through his active membership of the department's management team. He had his personal way to get things done,[3] which I try to capture below.

EV1: *Play the long game*
Science is for eternity. I don't think I ever heard Eelco say this explicitly, but in all his actions he made it clear he considered this evident. Consequently, it is an obligation and a calling to deliver the best quality research possible, in terms of results and presentation. It also means that we must take as much time as needed to discuss, sharpen, and truly understand each other's research results. It explains why Eelco had the stamina to work on long term projects, such as the (re)design of the Syntax Definition Formalism SDF (25 years, [12, 2]), the WebDSL language and system for web engineering (20 years [8, 13]), or the award winning[4] Spoofax language workbench (15 years, [9, 14]). On me and many others, Eelco's unshakable belief in the longevity of his endeavors had a magnetic effect. In 2000 (this was before Wikipedia existed), I gladly joined his grand ambition to unify all knowledge on *Program Transformation* in a new wiki program-transformation.org [4].

EV2: *Articulate a bold vision*
You can only play the long game if you know what you want the future to look like. Again, I haven't heard Eelco use the word "vision" often, but Eelco was always able to articulate the long term ideal world. He made it clear that he expected computer science professors to be programming; that computer scientists need the best possible equipment; and that researchers need excellent housing enabling them to do concentrated work. He also envisioned a computer science bachelor program with a substantial amount of theory (despite some TU Delft resistance). He explained why programming education needs online learning tools beyond the standard Integrated Development Environment. And that the research community needs a memory of conference activities, and can save valuable time and costs by relying on a single shared conference management system. On all important matters, Eelco would have a vision ready at hand.

[2] `https://eelcovisser.org/blog/2013/06/14/antoni-van-leeuwenhoek/`

[3] I'm pretty sure Eelco also lectured me on David Allen's "Getting Things Done" methodology, but I don't remember whether he followed it.

[4] `https://eelcovisser.org/blog/2021/02/08/spoofax-mip/`

EV3: *Just do it*

While some may think having a bold vision that is hard to realize is of little use, Eelco was committed to his ideals. He just started doing what was necessary. If the vision involved software, his preferred approach would be to start programming himself. If he needed others, he would use his magnetism to convince them to participate. Short of money, he would try to start making expenses anyway, trusting he would get forgiveness easier than permission. And, most importantly, he would work towards early successes that would get people addicted, such as a first version of the WebLab online programming education system [10]. Once addicted, the department could not say no to requests for resources anymore.

EV4: *Take resistance as encouragement*

Eelco's mission was to bring change, not just to TU Delft, but to computer science at large. If you advocate change, resistance is to be expected. To Eelco, this was inherent to academic life. I believe he considered it a good sign, confirming that he was indeed trying to change the status quo. Eelco was always willing, even eager, to engage in debate and listen to arguments, in his calm and friendly way. But resistance alone would be confirmation that he was onto something, rather than a reason to change course.

EV5: *Embrace education*

A key factor in Eelco's success at TU Delft was his dedication to education. This came from deep within: He was truly devoted to sharing his love for computer science in general and programming in particular. Eelco would use any opportunity to increase the teaching load of the Programming Language Group. Whenever we were searching for a teacher for a course, he would volunteer. This included undergrad courses, such as algorithms and data structures, which he would use to develop the WebLab infrastructure. He would involve his postdocs in the teaching, giving them useful experience for their later academic careers. And, Eelco would use it as an argument to justify growth of the Programming Languages Group: "Our high teaching load forces us to let postdocs teach: we need more faculty members." An argument that we happily subscribed to.

EV6: *Align individual and departmental interests*

Eelco ensured his Programming Languages Group fared well, for example in terms of office space, equipment, traveling, support staff, and starting packages. In the fight for (scarce) resources, this is not as obvious as it may sound. Eelco always made it clear that he wanted improvements for his group based on general principles that should hold for *everyone*. He would volunteer to help realize his vision, trusting that when successful his group would also reap the benefits.

EV7: *Be supportive*

Eelco was well aware that there is a cost to his approach to academic life. As a full professor with plenty of responsibilities, it takes resolve to create time for programming and individual research. Resistance always leaves a mark. Persistence takes energy, as does boldness. I believe this was also why Eelco was always ready to listen to his students, group members, or peers when they needed support. He related to the struggles and doubts, and tried to help where he could. He was there for me, too. Besides formal meetings, we regularly chatted about our own lives and careers, as well as the ups and downs of the department. On departmental matters we did not always agree. But I always knew that no matter what I would do, he would be there to support me. He had my back, and not just mine.

4 Thank You Eelco

There is much more that can be said about Eelco's footprint. Here I attempted to highlight his unique style and personality, through which he made a lasting impact on computer science research and education, at Delft University of Technology as well as in the international research community. Thank you Eelco for all you've done for us – we miss you very much.

References

1 Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Proc. 10th Int. Conf. on Compiler Construction (CC 2001)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001. `doi:10.1016/S1571-0661(04)80917-4`.

2 Luis Eduardo de Souza Amorim and Eelco Visser. Multi-purpose syntax definition with SDF3. In *18th International Conference on Software Engineering and Formal Methods (SEFM 2020)*, volume 12310 of *LNCS*, pages 1–23. Springer, 2020. `doi:10.1007/978-3-030-58768-0_1`.

3 Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

4 Arie van Deursen and Eelco Visser. The Reengineering Wiki. In *6th European Conference on Software Maintenance and Reengineering (CSMR 2002), 11-13 March 2002, Budapest, Hungary, Proceedings*, pages 217–220. IEEE Computer Society, 2002. `doi:10.1109/CSMR.2002.995808`.

5 Arie van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In *Proc. 1st Int. Workshop on Model-Driven Software Evolution*, pages 41–49, 2007. URL: `http://www.sciences.univ-nantes.fr/MoDSE2007/p19.pdf`.

6 Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*, pages 79–92. USENIX, 2004. URL: `http://www.usenix.org/publications/library/proceedings/lisa04/tech/dolstra.html`.

7 Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *26th ACM/IEEE Int. Conf. on Software Engineering (ICSE 2004)*, pages 583–592, 2004. `doi:10.1109/ICSE.2004.1317480`.

8 Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL: a domain-specific language for dynamic web applications. In *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, pages 779–780. ACM, 2008. `doi:10.1145/1449814.1449858`.

9 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proc. 25th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2010. `doi:10.1145/1869459.1869497`.

10 Tim van der Lippe, Thomas Smith, Daniël A. A. Pelsmaeker, and Eelco Visser. A scalable infrastructure for teaching concepts of programming languages in Scala with WebLab: an experience report. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016*, pages 65–74. ACM, 2016. `doi:10.1145/2998392.2998402`.

11 William Shadish, Thomas Cook, and Donald Cambell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, Boston, 2002.

12 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

13 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II (GTTSE) 2007*, volume 5235 of *LNCS*, pages 291–373. Springer, 2007. `doi:10.1007/978-3-540-88643-3_7`.

14 Guido Wachsmuth, Gabriël Konat, and Eelco Visser. Language design with the Spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014. `doi:10.1109/MS.2014.100`.

Renamingless Capture-Avoiding Substitution for Definitional Interpreters

Casper Bach Poulsen
Delft University of Technology, The Netherlands

Abstract

Substitution is a common and popular approach to implementing name binding in definitional interpreters. A common pitfall of implementing substitution functions is *variable capture*. The traditional approach to avoiding variable capture is to rename variables. However, traditional renaming makes for an inefficient interpretation strategy. Furthermore, for applications where partially-interpreted terms are user facing it can be confusing if names in uninterpreted parts of the program have been changed. In this paper we explore two techniques for implementing capture avoiding substitution in definitional interpreters to avoid renaming.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases Capture-avoiding substitution, lambda calculus, definitional interpreter

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.2

Supplementary Material *Software (Source Code)*: `https://github.com/casperbp/renamingless-capture-avoiding`; archived at `swh:1:dir:adec4695889aaeb0be4a180ed204d40fb55cf9ef`

Acknowledgements Thanks to Jan Friso Groote for feedback on a previous version of this paper, and to the anonymous reviewers who pointed out the work of Berkling and Fehr as an alternative solution to renamingless substitution.

1 Introduction

Following Reynolds [22], a definitional interpreter is an important and frequently used method of defining a programming language, by giving an interpreter for the language that is written in a second, hopefully better understood language. The method is widely used both for programming language research [3, 4, 13, 19, 23] and teaching [15, 20, 24]. A commonly used approach to defining name binding in such interpreters is *substitution*. A key stumbling block when implementing substitution is how to deal with *name capture*. The issue is illustrated by the following untyped λ term:

$$(\lambda f.\, \lambda y.\, (f\ \ 1) + y)\ (\lambda z.\, \underbrace{\ y\ }_{\text{free variable}})\ 2 \tag{1}$$

This term does *not* evaluate to a number value because y is a *free variable*; i.e., it is not bound by an enclosing λ term. However, using a naïve, non capture avoiding substitution strategy to normalize the term would cause f to be substituted to yield an interpreter state corresponding to the following (wrong) intermediate term $(\lambda y.\, ((\lambda z.\, y)\ \ 1) + y)\ 2$ where the red y is *captured*; that is, it is no longer a free variable.

Following, e.g., Curry and Feys [12], Plotkin [21], or Barendregt [5], the common technique to avoid such name capture is to *rename* variables either before or during substitution (a process known as α*-conversion* [11]). For example, by renaming the λ bound variable y to r, we can correctly reduce term (1) to $(\lambda r.\, ((\lambda z.\, y)\ \ 1) + r)\ 2$.

While a renaming based substitution strategy provides a well behaved and versatile approach to avoiding name capture, it has some trade-offs. For example, since renaming typically works by traversing terms, interpreters that rename at run time are typically slow. Furthermore, renaming gives intermediate terms whose names differ from the names in source programs. For applications where intermediate terms are user facing (e.g., in error messages, or in systems based on rewriting) this can be confusing. For this reason, interpreters often

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 2; pp. 2:1–2:10
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

use alternative techniques for (lazy) capture avoiding substitution, such as *closures* [16], *De Bruijn indices* [14], *explicit substitutions* [1], *locally nameless* [9]. However, traditional named variable substitution is sometimes preferred because of its simple and direct nature.

This paper explores named substitution strategies that do not rename variables. We explore two such strategies. The first technique we explore is a technique that Eelco Visser and I were using to teach students about static scoping, by having students implement definitional interpreters. To this end, we used a simple renamingless substitution strategy which (for applications that do not perform evaluation under binders) is capture avoiding. The idea is to delimit and never substitute into those terms in abstract syntax trees (ASTs) where all substitutions that were supposed to be applied to the term, have been applied; e.g., terms that have been computed to normal form. For example, using $\lfloor$ and $\rfloor$ for this delimiter, an intermediate reduct of the term labeled (1) above is $(\lambda y.\, (\ \lfloor(\lambda z.\, y)\rfloor\ \ 1) + y)\ 2$. Here the delimited highlighted term is closed under substitution, such that the substitution of y for 2 is not propagated past the delimiter; i.e., using $\rightsquigarrow$ to denote step-wise evaluation:

$$\begin{array}{ll}
 & (\lambda f.\, \lambda y.\, (f\ 1) + y)\ (\lambda z.\, y)\ 2 \\
\rightsquigarrow & (\lambda y.\, (\ \lfloor(\lambda z.\, y)\rfloor\ \ 1) + y)\ 2 \\
\rightsquigarrow & (\ \lfloor(\lambda z.\, y)\rfloor\ \ 1) + 2 \\
\rightsquigarrow & ((\lambda z.\, y)\ 1) + 2 \\
\rightsquigarrow & y + 2
\end{array}$$

The result term computed by these reduction steps is equivalent to using a renaming based substitution function. However, the renamingless substitution strategy we used does not rename variables (and so preserves the names of bound variables in programs), is simple to implement, and is more efficient than interpreters that rename variables at run time. A limitation of the renamingless substitution strategy is that it is not well-behaved for reduction strategies that perform evaluation under binders. That is, the strategy assumes that we treat any function expression $\lambda x.e$ as a value, such that the expression e is only ever evaluated if we apply the function. While this limits the applicability of the strategy, many operational semantics of λs adhere to this restriction [16, 18, 20, 22], including the ones we considered in the course we taught together. I never had the chance to discuss the novelty of the technique with Eelco. However, the technique we used in the course does not seem widely known or used. In this paper we explain and explore the technique and its limitations.

The second technique for capture-avoiding named substitution that we explore is an existing technique by Berkling and Fehr [7] which we were made aware of by a reviewer of a previous version of this paper. The technique has similar benefits as the technique we used in our course: it does not rename variables and is more efficient than interpreters that do renaming at run time. Furthermore, the technique does not make assumptions on behalf of interpretation strategy, and it supports evaluation under binders. On the other hand, Berkling and Fehr's substitution technique is more involved to implement and is a little less efficient than the renamingless substitution strategy that Eelco and I used in our course.

The renamingless techniques we consider in this paper are not new (at least the second technique is not; we do not expect that the first one is either, though we have not found it in the literature). But we believe they deserve to be more widely known. Our contributions are:

- We describe (§ 2) a simple, renamingless substitution technique for languages with open terms where evaluation does not happen under binders. The meta-theory of this technique is left for future work. We discuss and illustrate known limitations in terms of examples.
- We describe (§ 3) an existing and more general technique [7] which has similar benefits and does not suffer from the same limitations. However, its implementation is a little more involved to implement than the simple renamingless substitution strategy in § 2, and it is a little less efficient.

This paper is a literate Haskell document, available at `https://github.com/casperbp/renamingless-capture-avoiding`, and is structured as follows. § 2 describes a simple renamingless capture avoiding substitution strategy and its known limitations and § 3 describes Berkling-Fehr substitution which has similar benefits and fewer limitations but is less simple to implement. § 4 discusses related work and § 5 concludes.

2 Renamingless Capture-Avoiding Substitution

We present a simple technique for capture avoiding substitution, which avoids the need to rename bound variables. To demonstrate that the technique is about as simple to implement as substitution for closed terms (i.e., terms with no free variables, for which variable capture is not a problem), we first implement a standard substitution-based definitional interpreter for a language with closed, call-by-value λ expressions.

2.1 Interpreting Closed Expressions

Below left is a data type for the abstract syntax of a language with λs, variables, applications, and numbers. On the right is the substitution function for the language. The function binds three parameters: (1) the variable name (*String*) to be substituted, (2) the expression the name should be replaced by, and (3) the expression in which substitution happens.

```
data Expr0
  = Lam0 String Expr0
  | Var0 String
  | App0 Expr0 Expr0
  | Num0 Int
```

```
subst0 :: String → Expr0 → Expr0 → Expr0
subst0 x s (Lam0 y e)   | x ≡ y     = Lam0 y e
                        | otherwise = Lam0 y (subst0 x s e)
subst0 x s (Var0 y)     | x ≡ y     = s
                        | otherwise = Var0 y
subst0 x s (App0 e1 e2) = App0 (subst0 x s e1) (subst0 x s e2)
subst0 _ _ (Num0 z)     = Num0 z
```

The main interesting case is the case for Lam_0. There are two sub-cases, declared using *guards* (the Boolean expressions after the vertical bar). The first sub-case is when the variable being substituted matches the bound variable ($x \equiv y$). Since the inner variable shadows the outer, the substitution is not propagated into the body. In the other case (*otherwise*), the substitution is propagated. This other case relies on an implicit assumption that the expression being substituted by x does not have y as a free variable. If we violate this assumption, the substitution function and interpreter $interp_0$ on the left below is not going to be capture avoiding. Below right is an example invocation of the interpreter.

```
interp0 :: Expr0 → Expr0
interp0 (Lam0 x e)   = Lam0 x e
interp0 (Var0 _)     = error "Free variable"
interp0 (App0 e1 e2) = case interp0 e1 of
  Lam0 x e → interp0 (subst0 x (interp0 e2) e)
  _        → error "Bad application"
interp0 (Num0 z)     = Num0 z
```

```
> interp0 (App0 (Lam0 "x" (Var0 "x"))
                (Num0 42))
Num0 42
```

2.2 Intermezzo: Capture-Avoiding Substitution Using Renaming

The substitution function $subst_0$ relies on an implicit assumption that expressions are closed; i.e., do not contain free variables. If we want to support *open expressions* (i.e., expressions that may contain free variables), we must take care to avoid variable capture. A traditional approach [21] is to rename variables during interpretation, as implemented by the function $subst_{01}$ whose cases are the same as $subst_0$, except for the Lam_0 case:

```
subst01 x s (Lam0 y e) | x ≡ y      = Lam0 y e
                       | otherwise = let z = fresh x y s e
                                     in Lam0 z ( subst01 x s (subst01 y (Var0 z) e) )
```

Here *fresh x y s e* is a function that returns a fresh identifier if $x \notin FV(e)$ or $y \notin FV(s)$, or returns y otherwise. While this renaming based substitution strategy provides a relatively conceptually straightforward solution to the name capture problem, it requires an approach to generating fresh variables, and, since it performs two recursive calls to $subst_{01}$, it is inherently less efficient than the substitution function from § 2.1 – even in a lazy language like Haskell. Furthermore, depending on how *fresh* is implemented, the interpreter may not preserve the names of λ-bound variables. In the next section we introduce an simple alternative substitution strategy which does not rename or generate fresh variables, and which has similar efficiency as substitution for closed expressions.

2.3 Interpreting Open Expressions with Renamingless Substitution

Let us revisit the interpretation function $interp_0$ from § 2.1. Because our interpreter eagerly applies substitutions whenever it can, and because evaluation always happens at the top-level, never under binders, we know the following. Whenever the interpreter reaches an application expression $e_1\ e_2$, we know that *any variable that occurs free in* e_2 *corresponds to a variable that was free to begin with.* We can exploit this knowledge in our interpreter and substitution function. To this end, we introduce a dedicated expression form (the highlighted Clo_1 constructor below) which delimits expressions that have been closed under substitutions such that we never propagate substitutions past this closure delimiter:

```
data Expr1
  = Lam1 String Expr1
  | Var1 String
  | App1 Expr1 Expr1
  | Num1 Int
  | Clo1 Expr1
```

```
subst1 :: String → Expr1 → Expr1 → Expr1
subst1 x s (Lam1 y e)    | x ≡ y      = Lam1 y e
                         | otherwise = Lam1 y (subst1 x s e)
subst1 x s (Var1 y)      | x ≡ y      = s
                         | otherwise = Var1 y
subst1 x s (App1 e1 e2) = App1 (subst1 x s e1) (subst1 x s e2)
subst1 _ _ (Num1 z)     = Num1 z
subst1 _ _ ( Clo1 e)     = Clo1 e
```

Here $subst_1$ does not propagate substitutions into expressions delimited by Clo_1. The interpretation function $interp_1$ uses Clo_1 to close expressions before substituting (in the App_1 case), thereby avoiding name capture:

```
interp1 :: Expr1 → Expr1
interp1 (Lam1 x e)    = Lam1 x e
interp1 (Var1 x)      = Var1 x
interp1 (App1 e1 e2) = case interp1 e1 of
```

$$\begin{array}{lll}
\quad \mathit{Lam}_1\ x\ e \rightarrow \mathit{interp}_1\ (\mathit{subst}_1\ x\ (\mathit{Clo}_1\ (\mathit{interp}_1\ e_2))\ e) & & \\
\quad e'_1 \qquad\quad \rightarrow \mathit{App}_1\ e'_1\ (\mathit{interp}_1\ e_2) & & \\
\mathit{interp}_1\ (\mathit{Num}_1\ z) & = & \mathit{Num}_1\ z \\
\mathit{interp}_1\ (\mathit{Clo}_1\ e) & = & e
\end{array}$$

Whereas interp_0 explicitly crashes when encountering a free variable or when attempting to apply a non-function to a number, interp_1 may return a "stuck" term in case it encounters a free variable or an application expression that attempts to apply a value other than a function. The last case of interp_1 says that, when the interpreter encounters a closed expression, it "unpacks" the closure. This unpacking will not cause accidental capture: interpretation never happens under binders, so the only way the unpacked term can end up under a binder is via substitution. However, interp_1 only calls substitution on terms closed with Clo_1, thereby undoing the unpacking to prevent accidental capture.

To illustrate how interp_1 works, let us consider how to interpret $((\lambda f.\, \lambda y.\, f\ 0)\ (\lambda z.\, y)\ 1)$. The rewrites below informally illustrate the interpretation process, where for brevity we use λ notation instead of the corresponding constructors in Haskell and $\lfloor e \rfloor$ instead of $\mathit{Clo}_1\ e$:

$$\begin{array}{ll}
 & \mathit{interp}_1\ ((\lambda f.\, \lambda y.\, f\ 0)\ (\lambda z.\, y)\ 1) \\
\equiv & \mathit{interp}_1\ ((\lambda y.\, \lfloor(\lambda z.\, y)\rfloor\ 0)\ 1) \\
\equiv & \mathit{interp}_1\ (\lfloor(\lambda z.\, y)\rfloor\ 0) \\
\equiv & y
\end{array}$$

Unlike the renaming based substitution strategy discussed in § 2.2, our renamingless substitution strategy does not require renaming or generating fresh variables. Its efficiency is similar as substitution for closed expressions. It also preserves the names of binders. However, the renamingless substitution strategy in subst_1 and interp_1 relies on an assumption that evaluation does not happen under binders.

2.4 Limitation: Renamingless Substitution Does Not Support Evaluation Under Binders

The renamingless substitution strategy from § 2.3 assumes that terms under a Clo_1 have been closed under *all substitutions of variables bound in the context.* Interpretation strategies that evaluate under binders violate this assumption. For example, consider the interpreter given by $\mathit{normalize}_1$ whose highlighted recursive call performs evaluation under a λ binder:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{normalize}_1 :: \mathit{Expr}_1 \rightarrow \mathit{Expr}_1} \\
\mathit{normalize}_1\ (\mathit{Lam}_1\ x\ e) & = & \mathit{Lam}_1\ x\ (\mathit{normalize}_1\ e) \\
\mathit{normalize}_1\ (\mathit{Var}_1\ x) & = & \mathit{Var}_1\ x \\
\mathit{normalize}_1\ (\mathit{App}_1\ e_1\ e_2) & = & \textbf{case}\ \mathit{normalize}_1\ e_1\ \textbf{of} \\
\multicolumn{3}{l}{\quad \mathit{Lam}_1\ x\ e \rightarrow \mathit{normalize}_1\ (\mathit{subst}_1\ x\ (\mathit{Clo}_1\ (\mathit{normalize}_1\ e_2))\ e)} \\
\multicolumn{3}{l}{\quad e'_1 \qquad\quad \rightarrow \mathit{App}_1\ e'_1\ (\mathit{normalize}_1\ e_2)} \\
\mathit{normalize}_1\ (\mathit{Num}_1\ z) & = & \mathit{Num}_1\ z \\
\mathit{normalize}_1\ (\mathit{Clo}_1\ e) & = & e
\end{array}$$

Just like interp_1, the $\mathit{normalize}_1$ function closes off terms before substituting. However, because $\mathit{normalize}_1$ evaluates under λ binders, closures may be prematurely unpacked, which may result in variable capture. For example, say we apply $(\lambda x.\, \lambda y.\, x)$ to the free variable y. We would expect the result of evaluating this application to contain y as a free variable. However, using $\mathit{normalize}_1$, the free variable y is captured:

$$\begin{aligned}
&\mathit{normalize}_1\ ((\lambda x.\, \lambda y.\, x)\ y) \\
\equiv\ &\mathit{normalize}_1\ (\lambda y.\, \lfloor y \rfloor) \\
\equiv\ &\lambda y.\, \mathit{normalize}_1\ \lfloor y \rfloor \\
\equiv\ &\lambda y.\, y
\end{aligned}$$

The next section discusses a more general substitution strategy due to Berkling and Fehr [7] which does not have this limitation, which does not rename variables, and which is more efficient than the renaming based approach in § 2.2 but less efficient than the renamingless substitution strategy discussed in § 2.3.

3 Berkling-Fehr Substitution

Motivated by how to implement a functional programming language based on Church's λ-calculus [10], Berkling and Fehr [7] introduced a modified λ-calculus which uses a different kind of name binding and substitution. The key idea is to use a special operator (#) that acts on variables to neutralize the effect of one λ binding. For example, in the term $\lambda x.\, \lambda x.\, \#x$ the sub-term $\#x$ is a variable that references the *outermost* binding of x, whereas in $\lambda x.\, \lambda y.\, \#x$ the sub-term $\#x$ is a free variable.

Berkling and Fehr's # operator is related to De Bruijn indices [14] insofar as $\#^n x$ acts like an index that tells us to move n binders of x outwards. Indeed, if we were to restrict programs in Berkling and Fehr's calculus to use exactly one name, Berkling-Fehr substitution coincides with De Bruijn substitution. However, whereas De Bruijn indices can be notoriously difficult for humans to read (especially for beginners), Berkling-Fehr uses named variables such that indices only appear for substitutions that would otherwise have variable capture. This makes Berkling-Fehr variables easier to read for humans.

The definitions of shifting and substitution which we summarize in this section are taken from the work Berkling and Fehr [7] with virtually no changes. However, the language we implement is slightly different: they implement a modified λ-calculus with a call-by-name semantics, whereas we implement the same call-by-value language as in § 2. Our purpose of replicating their work is two-fold: to increase the awareness of Berkling-Fehr substitution and its seemingly nice properties, and to facilitate comparison with the renamingless approach we presented in § 2.3.

3.1 Interpreting Open Expressions with Berkling-Fehr Substitution

Below (left) is a syntax for λ expressions similarly to earlier, but now with Berkling-Fehr indices (right) instead of variables, where *Nat* is the type of natural numbers:

```
data Expr₂
  = Lam₂ String Expr₂
  | Var₂ Index
  | App₂ Expr₂ Expr₂
  | Num₂ Int
```

```
data Index = I {depth :: Nat, name :: String}
```

Here the (record) data constructor $I\ n\ x$ corresponds to an n-ary application of the special # operator to the name x; i.e., $\#^n x$. We will refer to the n in $I\ n\ x$ as the *depth* of an index. As discussed above, a Berkling-Fehr index is similar to a De Bruijn index except that whereas a De Bruijn index tells us how many scopes to move out in order to locate a binder, a Berkling-Fehr index tells us how many scopes *that bind the same name* to move out in order to locate a binder. In what follows, we will sometimes use λ notation as informal syntactic sugar for the constructors in Haskell above. When doing so, we use "naked" variables x as informal syntactic sugar for a variable at depth 0; i.e., $\mathit{Var}_2\ (I\ 0\ x)$.

To define Berkling-Fehr substitution, we need a notion of *shifting*. Shifting is used when we propagate a substitution, say $x \mapsto e$ where x is a name and e is an expression, under a binder y. To this end, a shift increments the depth of all free occurrences of y in s by one. Such shifting guarantees that free occurrences of y in s are not accidentally captured.

$$\begin{array}{lll}
\mathit{shift} :: \mathit{Index} \to \mathit{Expr}_2 \to \mathit{Expr}_2 & & \\
\mathit{shift}\ i\ (\mathit{Lam}_2\ x\ e) & \mid \mathit{name}\ i \equiv x & = \mathit{Lam}_2\ x\ (\mathit{shift}\ (\mathit{inc}\ i)\ e) \\
 & \mid \mathit{otherwise} & = \mathit{Lam}_2\ x\ (\mathit{shift}\ i\ e) \\
\mathit{shift}\ i\ (\mathit{Var}_2\ i') & \mid \mathit{name}\ i \equiv \mathit{name}\ i' & \\
 & \quad \wedge\ \mathit{depth}\ i \leqslant \mathit{depth}\ i' & = \mathit{Var}_2\ (\mathit{inc}\ i') \\
 & \mid \mathit{otherwise} & = \mathit{Var}_2\ i' \\
\mathit{shift}\ i\ (\mathit{App}_2\ e_1\ e_2) & = \mathit{App}_2\ (\mathit{shift}\ i\ e_1)\ (\mathit{shift}\ i\ e_2) & \\
\mathit{shift}\ \_\ (\mathit{Num}_2\ z) & = \mathit{Num}_2\ z &
\end{array}$$

The *shift* function binds an index as its first argument. The name of this index (e.g., x) denotes the name to be shifted. The depth of the index denotes the *cut-off* for the shift; i.e., how many #'s an x must at least be prefixed by before it is a free variable reference to x. For example, say we wish to shift all free references to x in the term $\lambda x.\, x\ (\#x)$. We should only shift $\#x$, not x, since x references the locally λ bound x. For this reason, the shift function uses a cut-off which is incremented when we move under binders by the same name as we are trying to shift. For example:

$$\begin{array}{l}
\quad \mathit{shift}\ x\ (\lambda x.\, x\ (\#x)) \\
\equiv \lambda x.\, (\mathit{shift}\ (\#x)\ x)\ (\mathit{shift}\ (\#x)\ (\#x)) \\
\equiv \lambda x.\, x\ (\#\#x)
\end{array}$$

The Berkling-Fehr substitution function subst_2 applies shifting to avoid variable capture when propagating substitutions under λ binders:

$$\begin{array}{lll}
\mathit{subst}_2 :: \mathit{Index} \to \mathit{Expr}_2 \to \mathit{Expr}_2 \to \mathit{Expr}_2 & & \\
\mathit{subst}_2\ i\ s\ (\mathit{Lam}_2\ x\ e) & \mid \mathit{name}\ i \equiv x & = \mathit{Lam}_2\ x\ (\mathit{subst}_2\ (\mathit{inc}\ i)\ (\mathit{shift}\ (I\ 0\ x)\ s)\ e) \\
 & \mid \mathit{otherwise} & = \mathit{Lam}_2\ x\ (\mathit{subst}_2\ i\ (\mathit{shift}\ (I\ 0\ x)\ s)\ e) \\
\mathit{subst}_2\ i\ s\ (\mathit{Var}_2\ i') & \mid i \equiv i' & = s \\
 & \mid \mathit{otherwise} & = \mathit{Var}_2\ i' \\
\mathit{subst}_2\ i\ s\ (\mathit{App}_2\ e_1\ e_2) & = \mathit{App}_2\ (\mathit{subst}_2\ i\ s\ e_1)\ (\mathit{subst}_2\ i\ s\ e_2) & \\
\mathit{subst}_2\ \_\ \_\ (\mathit{Num}_2\ z) & = \mathit{Num}_2\ z &
\end{array}$$

To interpret an Expr_2 application $e_1\ e_2$, we first interpret e_1 to a function $\lambda x.\, e$, and then substitute x in the body e, such that occurrences of x at a higher depth are left untouched. But after we have substituted the bound occurrences of x in e, the depth of the remaining occurrences of x in e need to be decremented. To this end, we use an *unshift* function which decrements the depth of a given name, modulo a cut-off which now tells us what depth a name has to strictly be larger than in order for it to be a free variable to be unshifted:

```
unshift :: Index → Expr2 → Expr2
unshift i (Lam2 x e)   | name i ≡ x             = Lam2 x (unshift (inc i) e)
                       | otherwise              = Lam2 x (unshift i e)
unshift i (Var2 i′)    | name i ≡ name i′
                         ∧ depth i < depth i′ = Var2 (dec i′)
                       | otherwise              = Var2 i′
unshift i (App2 t1 t2) = App2 (unshift i t1) (unshift i t2)
unshift _ (Num2 z)     = Num2 z
```

Using *unshift*, we can now implement an interpreter that does evaluation under λs and that uses capture-avoiding substitution:

```
normalize2 :: Expr2 → Expr2
normalize2 (Lam2 x e)   = Lam2 x (normalize2 e)
normalize2 (Var2 i)     = Var2 i
normalize2 (App2 e1 e2) = case normalize2 e1 of
  Lam2 x e → unshift (I 0 x) (normalize2 (subst2 (I 0 x) (normalize2 e2) e))
  e1′      → App2 e1′ (normalize2 e2)
normalize2 (Num2 z)     = Num2 z
```

The problematic program from § 2.4 now yields a result with a free variable, as expected:

$$normalize_2\ ((\lambda x.\, \lambda y.\, x)\ y) \ \equiv\ \lambda y.\, \#y$$

3.2 Relation to Renamingless Substitution

On the surface, the techniques involved in Berkling-Fehr substitution and our renamingless substitution strategy from § 2 may seem different. A common point between the two is that they avoid renaming by strategically closing off certain variables to protect them from substitutions from lexically closer binders, and strategically reopening those variables to substitutions coming from lexically distant binders.

The renamingless substitution strategy achieves this by using a syntactic and rather coarse-grained discipline which closes entire sub-branches over all possible substitutions, similar to how closures à la Landin [16]. When the interpreter reaches a closed sub-expression, it is re-opened. As discussed, this discipline works well for languages that do not perform evaluation under binders. While we demonstrated the technique using a call-by-value language in § 2, the technique is equally applicable to call-by-name interpretation. But not for languages that perform evaluation under binders.

Berkling-Fehr substitution uses a more fine-grained approach to strategically close off variables to protect them from substitutions from lexically closer binders, by shifting free occurrences of variables when moving under a binder. When a binder is eliminated, terms are unshifted. This fine-grained approach is not subject to the same limitations as the renamingless approach from § 2.3. Indeed, in their paper, Berkling and Fehr [7] prove that their notion of substitution and their modified λ-calculus is consistent with Church's λ calculus. Since shifting and unshifting requires more recursion over terms than the simpler renamingless approach from § 2, Berkling-Fehr substitution is less efficient. However, it is still more efficient than the renaming approach discussed in § 2.2.

As discussed, Berkling-Fehr substitution is closely related to De Bruijn indices, the main difference being that Berkling-Fehr use names and are more readable. To work around the readability issue with De Bruijn indices, one might also combine a named and De Bruijn approach where variable nodes comprise *both* a name *and* a De Bruijn index. But that leaves the question of how to disambiguate programs with ambiguous name. For example, using this approach, how would the pretty-printed version of the Berkling-Fehr indexed expression $\lambda x.\, \lambda x.\, \# x$ look? Berkling-Fehr indices strike an attractive balance between efficiency, preserving names from source programs, and readability.

4 Related Work

In this paper we explored two techniques for capture avoiding substitution that avoids renaming, for the purpose of implementing static name binding in languages with λs. The topic of evaluating λ expressions has a long and rich history. Summarizing it all is beyond the scope of this paper; for overviews see, e.g., the works of Barendregt [6] or Cardone and Hindley [8]. We discuss a few of the papers that are most closely related to the techniques we have described.

In their formalization of λ calculus and type theory, McKinna and Pollack [17] consider a system that uses named substitution without renaming, for a particular notion of open terms. They consider a syntax that distinguishes two classes of names: *parameters* and *variables*. *Variable substitution* does not affect parameters, and *parameter substitution* does not affect variables. Their notion of variable substitution is defined for terms that are *variable-closed*, but which may be *parameter-open*. Thus, by encoding free variables as parameters, their system can be used to compute with open terms. However, syntactically distinguishing free variables this way seems to presupposes a static binding analysis. The approach we discussed in § 2.3 does not presuppose such static analysis.

Our paper considers how to interpret open terms. There exist several calculi in the literature for evaluating open terms. Accatolli and Guerrieri [2] gives an overview of several of these calculi for *open call-by-value*, which is the class of languages that the interpreters in § 2 and § 3 interpret. Accatolli and Guerrieri focus on the meta-theory of these calculi. To this end, they rely on an unspecified notion of capture-avoiding substitution. In this paper, we explore how to implement such capture-avoiding substitution functions in interpreters in a way that does not perform renaming.

5 Conclusion

We have discussed two techniques for implementing capture avoiding substitution in definitional interpreters in a way that does not require renaming of bound variables. One of the techniques relies on a coarse-grained but simple discipline for closing terms which works well for interpretation strategies that do not evaluate under binders. The other technique, due to Berkling and Fehr [7], is more fine-grained and also works for interpretation strategies that evaluate under binders. While less expressive, the former technique is simpler to implement, and is slightly more efficient. Neither of the two techniques seem to be widely known or at least not widely applied. With this work, we hope to increase awareness of these techniques.

References

1 Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. `doi:10.1017/S0956796800000186`.

2 Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226, 2016. `doi:10.1007/978-3-319-47958-3_12`.

3 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. `doi:10.1145/3093333.3009866`.

4 Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018. doi:10.1145/3158104.

5 Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

6 Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bull. Symb. Log.*, 3(2):181–215, 1997. doi:10.2307/421013.

7 K. J. Berkling and Fehr E. A modification of the λ-calculus as a base for functional programming languages. In *ICALP 1982*. Springer Berlin Heidelberg, 1982.

8 Felice Cardone and J. Roger Hindley. *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, chapter History of Lambda-calculus and Combinatory Logic. Elsevier, 2009.

9 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

10 Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

11 Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. doi:10.2307/2371045.

12 Haskell B. Curry and Robert Feys. *Combinatory Logic*. Combinatory Logic. North-Holland Publishing Company, 1958.

13 Nils Anders Danielsson. Operational semantics using the partiality monad. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 127–138. ACM, 2012. doi:10.1145/2364527.2364546.

14 N.G de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

15 Shriram Krishnamurthi. Programming languages: Application and interpretation. https://www.plai.org/3/2/PLAI%20Version%203.2.0%20electronic.pdf, 2002. Accessed: 2022-12-01.

16 P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi:10.1093/comjnl/6.4.308.

17 James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3-4):373–409, 1999. doi:10.1023/A:1006294005493.

18 Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

19 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-49498-1_23.

20 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

21 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

22 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. doi:10.1023/A:1010027404223.

23 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.

24 Jeremy Siek. *Essentials of Compilation*. MIT Press, 2022.

Injecting Language Workbench Technology into Mainstream Languages

Michael Ballantyne
PLT at Northeastern University, Boston, MA, USA

Matthias Felleisen
PLT at Northeastern University, Boston, MA, USA

Abstract

Eelco Visser envisioned a future where DSLs become a commonplace abstraction in software development. He took strides towards implementing this vision with the Spoofax language workbench. However, his vision is far from the mainstream of programming today. How will the many mainstream programmers encounter and adopt language workbench technology? We propose that the macro systems found in emerging industrial languages open a path towards delivering language workbenches as easy-to-adopt libraries. To develop the idea, we sketch an implementation of a language workbench as a macro-library atop Racket and identify the key features of the macro system needed to enable this evolution path.

2012 ACM Subject Classification Software and its engineering → Macro languages; Software and its engineering → Domain specific languages; Software and its engineering → Development frameworks and environments

Keywords and phrases Language workbenches, macro systems, language adoption

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.3

Supplementary Material
Software (Source Code): `https://github.com/michaelballantyne/syntax-spec`
archived at `swh:1:dir:1dd5436fae3756287949abb1df145f767aad1852`

Funding This work was partially supported by NSF grants SHF 2007686 and 1823244.

Acknowledgements The authors are grateful for technical feedback from Michael Delmonaco, Siddhartha Kasivajhula, and Alexis King. Also thanks to William Byrd and Siddhartha Kasivajhula for helpful comments on early drafts, and to Shriram Krishnamurthi and Ryan Culpepper for rubbing Matthias's nose in Spoofax and language workbench ideas.

1 Bringing Eelco Visser's legacy to the mainstream

Eelco Visser envisioned a future where creating a new programming language becomes a common means of abstraction. One of his well-known examples is that of a web programming DSL. In WebDSL [24], a programmer specifies only the data model, page flow, and page layouts. The DSL compiler generates all operational elements such as request handlers.

In order to implement such DSLs, Visser sought to provide programmers with a "language designer's workbench". A programmer specifies a DSL in a declarative manner, and the workbench derives compilers, IDEs, verifiers, and documentation [25]. The Spoofax Language Workbench [18] represents Visser's last step in a long chain of research accomplishments towards this vision.

Sadly, Visser did not see his vision spread to the mainstream of software development. We hypothesize one contributing cause. Spoofax requires teams to adopt language-building wholesale or not at all. A programmer must cease to be a Java programmer, and instead become a Spoofax programmer. Worse, the programmer's teammates must use a new IDE, a new build system, and possibly other tools.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 3; pp. 3:1–3:11
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We share Visser's broad goal and call it language-oriented programming [12,27]. In contrast to the vision behind Spoofax, our working hypothesis calls for an incremental evolution path from the current state of affairs to one where developers embrace the construction of DSLs. Programming technologies created by researchers most often reach the mainstream when they are adopted by existing industrial languages. Hence to achieve mainstream adoption, language-building technology must be integrated into existing language ecosystems.

This paper sketches one feasible evolution path by which mainstream programmers and languages can integrate language workbench ideas. This evolution requires removing hurdles to adoption at two levels. The first concerns friction at the moment an individual programmer chooses to use or create a DSL. If adopting a DSL or a language workbench requires installing and using new tools, the programmer may find the cost too high. Instead, a programmer should be able to begin using a DSL or creating a DSL as easily as using or creating a library. In a typical programming ecosystem, only the language itself is universally available to all programmers. Different programmers may use different IDEs and build tools. Thus to ensure any programmer can easily begin using and making DSLs, the language workbench must be made available as part of the language itself.

The second hurdle thus concerns the integration of DSL-building tools into a language. The creators of existing industrial languages are unlikely to integrate and standardize complex language workbench features before they are widely used. However, a number of widely-used languages including Rust, Scala, and Julia are extensible via macro systems. If language workbench features can be built as macro-libraries, they can be used as part of the language without being built-in to the core.

For the past year, we have investigated this path by building a language workbench as a macro library in Racket, a general-purpose language with a sophisticated macro system. In the process we have identified the essential macro system features needed to support a language workbench macro-library and those features that are superfluous. We anticipate that this experience can guide the design of extensions to macro systems in industrial languages and thus the development of language workbenches as libraries.

Our initial prototype realizes a part of Visser's ideal language designer's workbench. It accounts for only some aspects of language specification, and it does not yet generate rich IDE services or verification infrastructure. We expect that further development can close these gaps and make a great many of the technologies pioneered by Visser accessible in existing languages. However, some tradeoffs are fundamental to the approach. Our language workbench libraries give up generality in order to specialize to the conventions of their host. We also focus specifically on the case of DSLs intended for professional software engineers, whereas language workbenches are also used to create DSLs for domain experts and end-users.

The remainder of the paper proceeds as follows: Section 2 illustrates our objective by showing the experience of using and defining DSLs as libraries. Section 3 describes the implementation of the meta-DSL and the set of macro system features that are and are not needed to support it. Section 4 contrasts our work to related approaches, and suggests further ideas that we may adapt from language workbenches in the future. Section 5 anticipates the issues that need to be addressed to create language workbench macro-libraries for other host languages. The last section provides a summary and an outlook.

2 An integrated language workbench

The best way to understand how developers should be able to use workbenches inside their ecosystem is to work through an example. Consider a programmer who is implementing a user interface that loads and displays a CSV file from a remote server. The controller for the UI component is easily understood as a state machine. The states include the initial

```
#lang racket

(require state-machine)

;; GUI elements and data processing code in Racket (simplified)
(define table (new list-box%))
(define url-field (new text-field%))
(define load-button
  (new button% [callback (lambda _ (send csv-controller load-click))]))
(define (load-data url)
  ;; elided code to load CSV from the URL
  (send csv-controller loaded data))

;; Controller
(define csv-controller
  (machine
   #:initial-state no-data
   (state no-data
     (on-enter (set-display url-message)))
   (state loading
     (on-enter (set-display loading-message)
               (load-data (send url-field get-value)))
     (on (loaded data)
       (set-data data)
       (-> display)))
   (state display
     (on-enter (set-display table)))
   (on (load-click)
     (-> loading))))
```

Figure 1 CSV browser UI with a controller implemented using a state machine DSL.

state, before a URL has been entered; the loading state; and the final state where the table is displayed. Asynchronous user actions and network events drive the transitions between states.

In a conventional programming language the programmer must map these concepts to language elements such as classes and methods. The programmer might create a class representing the machine, an interface for states including methods for each transition, and classes implementing the interface for each state. The original machine structure easily becomes swamped by implementation details, and it may be difficult to decipher when returning to the code sometime later.

2.1 Using a DSL as a library

In contrast, writing the controller using a state machine DSL directly expresses the structure the programmer has in mind. Figure 1 shows a program using a state machine DSL implemented with our language-workbench library. It uses the DSL alongside general purpose GUI and data processing code written in Racket. The library import `(require state-machine)` makes the DSL available in the module. Subexpressions written in the host language (here Racket, displayed in lighter font) integrate the state machine with the GUI code. The state machine DSL's compiler generates code that uses Racket's object-oriented programming facilities to implement the state machine with classes for states and methods for transitions. Racket code that triggers machine transitions interacts with the controller as an object using Racket's method call form `send`. Thus the generated implementation is much the same as the programmer would write without access to a DSL, but the mapping from domain concepts to host-language constructs is encapsulated in the DSL implementation.

The ease with which a programmer can integrate this DSL code is essential to its value. A programmer is likely to use this approach if it merely means importing a library. However, a programmer is unlikely to find the benefit sufficient if it requires a major change in workflow such as a new IDE or build system component. Thus a platform for programming with DSLs should minimize the cost of integrating new languages into a program. Racket's "languages as libraries" [22] and SugarJ's "sugar libraries" [11] realize this goal.

2.2 Implementing a DSL as a library

Of course, someone must first have done the work to implement the state machine language. The meta-DSLs offered by language workbenches promise to simplify this task. Our meta-DSL, integrated with Racket, aims to provide the advantages of a language workbench without requiring a programmer to leave the familiar host language. Each aspect of our meta-DSL design aims at making it easy for programmers familiar with Racket to understand and adopt.

Figure 2 shows how the state machine language is defined in our meta-DSL.[1] Like other DSLs in Racket, the meta-DSL is a "language as a library", so it is used in a normal Racket module via the `(require syntax-spec)` declaration. Similarly, the `machine` syntax is exported with `provide` in the same way as any normal function or value definition. Thus implementing a DSL involves no more friction than implementing a library.

The state machine language definition specifies syntax via grammar nonterminals and name binding via binding rules associated with nonterminal productions. The definition of `machine` establishes the interface between the state machine DSL and the host language. It includes a call to a back-end compiler for the DSL, `compile-machine`, which is implemented in compile-time Racket code.

We choose to account for syntax, binding rules, and the interface with the host language in our meta-DSL. These portions of a language implementation are the most uniform across DSLs, because their structure relates as closely to the host language as to the domain of the language being defined. This connection also means that these elements require the most intricate interaction with the host language implementation. The underlying implementation must interact with the host's data representations of syntax, scope, and binding environments. In an extensible language such as Racket, the processes of parsing and name resolution interleave in complex ways. Thus a DSL implementation must perform operations in the correct order to ensure that data is available when needed. Hiding these elements via the meta-DSL means that programmers need not be aware of these effectful operational details. We leave programmers to implement the back-end compilation to Racket using conventional procedural Racket code augmented by the existing `syntax-parse` pattern matching and templating DSL [8].

[1] The meta-DSL is available at `https://github.com/michaelballantyne/syntax-spec`, and the state machine example at `https://github.com/michaelballantyne/syntax-spec/tree/main/demos/visser-symposium`.

```
#lang racket

(provide machine)
(require syntax-spec)

(syntax-spec
  (binding-class state-name)

  (host-interface/expression
    (machine #:initial-state s:state-name d:machine-decl ...)
    #:binding {(recursive d) s}
    (compile-machine #’s #’(d ...)))

  (nonterminal/two-pass machine-decl
    (state n:state-name
      e:event-decl ...)
    #:binding (export n)
    e:event-decl)

  (nonterminal event-decl
    (on-enter e:racket-expr ...)
    (on (evt:id arg:racket-var ...)
      e:racket-expr ...
      ((~datum ->) s:state-name))
    #:binding {(bind arg) e}))
```

Figure 2 Syntax and binding rules declaration for the state machine DSL.

Our meta-DSL is optimized for specifying syntaxes and binding structures similar to those of Racket itself. This restriction helps programmers understand programs in the meta-DSL because of the programmers’ knowledge of the host language. Furthermore, restricting the expressivity of the meta-DSL means that the DSLs created with it share a common structure, so users of a collection of such DSLs may develop transferrable intuitions. Following this design idea, DSLs created in our framework re-use the S-expression syntax of Racket. They feature tree-structured scope and binding, with a two-pass operational model of name analysis and macro expansion. The DSL front-end defined in the meta-DSL checks the same degree of static semantics as in Racket: syntax and name binding. Any other static semantics must be checked in subsequent passes of the DSL compiler.

3 A language workbench as a macro-library

Our meta-DSL is implemented via procedural macros that transform the specification of a DSL into macros that process DSL programs. Concretely, the `syntax-spec` macro generates compile-time datatypes and analysis functions corresponding to binding categories and nonterminals. The analysis functions traverse the DSL syntax and create a representation of scopes and name bindings. The `host-interface/expression` syntax generates a macro that serves as the entry point into a DSL implementation. The state machine language’s `machine` is an example of such a macro. The generated macro calls the DSL’s analysis functions. It then invokes the programmer-defined DSL compiler on the result of this analysis to obtain host language code, which the macro returns as its expansion.

3.1 Essential macro system features

Our approach relies on a host language with a procedural macro system sporting a basic set of features:

- Macros consume a data representation that is flexible enough to support DSL syntaxes.
- Macros are capable of generating further procedural macros and other compile-time code.
- Macros can generate fresh names and names that reliably reference exports of a given library.

A variety of languages have macro systems that meet this most basic set of requirements: Common Lisp, R6RS Scheme, Clojure, Rust, Scala, Template Haskell, Julia, and Elixir.

Less commonly available macro system features are required to support other aspects of language workbenches. To allow fine-grained intermixing of host-language code within DSL syntax, the host macro system needs to come with a reflective API. The analysis function for a DSL nonterminal must be able to (1) determine whether a name has a meaning established in the surrounding context; (2) create an extended binding environment with entries for new names; and (3) analyze a host language subexpression in such an extended environment. In Racket, definition contexts, `local-expand`, and manipulation of scope sets provide these capabilities [15, 16]. Our implementation builds on top of an API that provides higher-level abstractions over these concepts [2].

Finally, to allow parts of DSL programs to be written in separately compiled modules, the host macro system must provide a means to persist a compile-time environment across separate compilations. Racket's notion of "visits" allows macros to leave behind expressions that are re-evaluated every time one module is loaded to support the analysis of another [14]. Such expressions may construct compile-time tables associating name bindings with arbitrary values.

3.2 The full power of Racket's macros is not needed

Interestingly, our experience suggests that some of the most-heralded features of Racket's macro system are *not* essential to support a language-workbench library.

The binding specifications provided as part of nonterminal definitions would allow the analysis functions to implement name hygiene on top of macro systems with relatively naive macro hygiene. The form of hygiene available in Clojure, for example, would suffice. The more sophisticated macro hygiene found in Scheme and Racket that infers the intended scoping structure from the expansion of a macro is unnecessary.

Similarly, the careful phase-separation and separate compilation guarantees provided by Racket's macro system are essentially unnecessary. The meta-DSL implementation handles the symbol table operations performed during analysis. Because all the needed side-effects occur within the meta-DSL implementation, there is less opportunity for programmer error in DSL implementations to create effect dependencies that break separate compilation.

Even though the reflective APIs discussed above are critical to support our meta-DSL, they are only needed in restricted form. For example, DSL analysis functions need to invoke the host analyzer on subexpressions but do not need to examine the results; opaque values would suffice. Likewise, the analysis functions need a way of associating compile-time information with names, but this could take a simpler form than Racket's support for visit-time evaluation with managed side-effects.

4 Related approaches

4.1 Internal and embedded DSLs

DSLs created with our language-workbench library might at first appear similar to internal or embedded DSLs. However, both of those terms tend to refer to DSLs where the DSL code not only appears within host-language syntax but is encoded in the host language syntax and semantics. For example, the syntax of an embedded state machine DSL might consist of host-language function calls to methods with names such as “state” and “on-event”. Embedded DSL semantics are defined in one of two ways [17]. In *shallow* embeddings, the evaluation of the embedding host-language code immediately realizes the DSL evaluation. In *deep* embeddings, the evaluation of host-language code that embeds DSL code first produces a data representation of DSL abstract syntax, which is then interpreted or compiled. Thus, in deep embedding a DSL compiler can perform any kind of static checking or optimization, but it all takes place at run time of the host language.

In contrast, our style of DSLs use only the lexical or reader syntax of the host language. Their syntax and semantics are defined by custom compiler components (parser, static checker, code generator) that run at host-language compile time. Running as part of host-language compilation means that any static errors are raised at compile time in the IDE.

4.2 Language workbenches

Our meta-DSL adapts ideas from the language workbench tradition into a new context where we anticipate an opportunity for broad adoption.

Spoofax aims to be general, with meta-DSLs expressive enough to implement the full gamut of designs compatible with well-established programming language theory. This principled, research-focused approach pushes the expressivity of language workbench meta-DSLs. That expressivity comes with a cost, however: programmers need to learn an expansive language and have some familiarity with the underlying theory. To reduce this cost and flatten the learning curve we focus instead on the restricted scenario of building DSLs that fit together on top of a single host language.

SugarJ represents an alternative approach to integrating language workbench meta-DSLs with a host language [11]. It provides a front-end that integrates the syntax of Java together with (predecessor versions of) Spoofax meta-DSLs for syntax and transformation. While SugarJ achieves linguistic integration, it requires a non-standard compiler. It is not as easy to adopt as a library, which is the key insight for our approach.

4.2.1 IDE services

Language workbenches such as Spoofax and SugarJ generate more from a DSL definition than just a compiler. They also automatically generate IDE services. These services range from basic syntax coloring and bracket matching to rich semantic services. For example, Spoofax generates code completion that takes into account the syntax, static semantics, and name binding of the language [20]. With additional effort DSL creators can also implement custom transformations such as refactorings.

DSLs created with our prototype meta-DSL integrate with the DrRacket IDE in the same manner as other macro-based DSLs in Racket [13]. However, the provided services are limited to error highlighting, jump-to-definition, and rename refactorings. Macro-extensible languages have long faced challenges providing advanced IDE services. Syntax defined by a macro is specified only via its procedural expansion into host-language syntax. This lack of structure makes it difficult to implement parsing, name analysis, or typechecking processes that recover from errors. Autocompletion faces a similar problem taking into account the grammar and static semantics of a macro-defined DSL.

Looking ahead, the syntax and binding rule declarations in our meta-DSL provide the structure that is missing from conventional macro definitions. Thus it should be possible to derive rich IDE services for DSLs specified in our language-workbench library. The open challenge lies in connecting the editor-service code that would be generated by the meta-DSL to the IDE. One possible approach is to modify the interface for macros to separate out analysis and compilation procedures. The analysis procedure would return the information needed to implement IDE services. The host language could then expose these services via the Language Server Protocol [4, 7]. While the task of implementing such an analysis interface for each language extension would be a burden for a traditional macro author, the meta-DSL can generate it automatically.

4.2.2 DSLs for domain experts

Some DSLs are designed with non-programmer domain experts in mind, rather than software engineers. Implementing a DSL for non-programmers demands different design considerations than those addressed by our meta-DSL. Fluid integration with general-purpose code and standard programming tools is less critical. On the other hand, providing structured editing to reduce errors, guidance via autocompletion or form-like interfaces, and live reaction to edits to show their effects become more important. Visual representations of domain concepts can also make their manipulation more tangible. Thus language workbenches such as JetBrains MPS [26] that produce external DSLs equipped with projectional editors provide advantages for these situations.

Nonetheless, our language-workbench library may sometimes be a good way to create DSLs for domain experts. Especially for initial prototyping, the ease with which a library-based DSL integrates into an existing software project may present an attractive tradeoff against the advantages of a more sophisticated implementation in a standalone workbench. As we improve the IDE services provided by our language-workbench library, this tradeoff becomes more advantageous. Furthermore, several directions of prior work suggest ways to integrate the custom editor services and visual and interactive elements that are important for domain experts. “Editor libraries” allow DSLs to provide custom editor services such as refactorings [10]. “Visual syntax” provides a mechanism for using interactive visualizations in place of textual syntax for macro-defined language extensions [1]. Along similar lines, “livelits” provide interactive syntactic elements that take on a more limited linguistic role but integrate with live evaluation [19].

5 Scaling up to mainstream host languages

Our meta-DSL design is specialized to Racket’s conventions for syntax and static semantics. When applying the idea in another host language these design elements would vary.

Furthermore, Racket is designed with extensibility in mind. S-expression syntax makes language extension within that syntax easy, and Racket features a module system designed specially to support macros and macro libraries [14]. Racket thus has all the macro system features we need to host a language workbench as a library. This makes Racket a good platform for prototyping our idea, but also the easiest case.

We anticipate that replicating our meta-DSL in another dynamically typed, Lisp-family language such as Clojure would be relatively easy. Clojure features S-expressions, procedural macros, a form of macro hygiene, and reflection on the compile-time environment. Small extensions to the macro system would be needed to allow macros to record data about DSL variables in the compile-time environment and to invoke Clojure's macro expander in an extended environment.

Applying the idea in a language such as Rust presents a greater, yet surmountable challenge. Like Clojure, Rust supports procedural macros, includes macro hygiene, and uses a representation of syntax that is rich enough to support interesting DSL syntax [21]. Extensions similar to those discussed above for Clojure would suffice for implementing a basic language workbench library. In the context of Rust, these APIs need to be made type-safe and the added compile-time information needs to be plumbed between separate compilations.

In a typed language like Rust programmers would benefit substantially from integration between the type systems of each DSL and the host language. Ideally a language-workbench library for Rust would feature a meta-DSL like Statix [23] for specifying DSL type systems, suitably modified to integrate with Rust's type system. Unfortunately, Rust typechecking occurs only after macro expansion, meaning that type information is not available to macros, and any type errors are currently reported in terms of expanded code. A macro API that provides the right kind of interaction with the host language type system is an open research problem. Previous research on integrating macro expansion and typechecking in the Turnstile meta-DSL for Racket [5, 6] and the Klister language [3] suggest potential directions.

Unfortunately the metaprogramming systems present in many of today's languages are inadequate to properly host a language-workbench library. For example, Java's annotation processors [9] cannot introduce new syntax. Annotation arguments can only be simple values or arrays. Annotation processors also cannot change the way the processed class compiles; they can only generate additional classes. If our approach succeeds in those languages that currently have capable macro systems, we hope that this success will inspire the creators of other mainstream languages to add support for macros.

6 Conclusion

This paper has outlined what we consider a feasible evolution path for introducing language workbench technologies into an existing, mainstream programing language. Integrating a language workbench meta-DSL into an existing language ecosystem reduces friction at the moment a programmer chooses to adopt or create a DSL. Tailoring the design of the meta-DSL to use concepts familiar from the host language narrows the gap between programmers' existing knowledge and DSL creation. Implementing the meta-DSL as a macro library minimizes the additions needed to the core of a host language to support the approach.

Our experience from prototyping such a meta-DSL in Racket revealed that certain macro system features missing from mainstream languages are necessary to support a language workbench. However, we also found that the most complex features of Racket's macro system are not essential for this application. Given the additional information afforded by a declarative specification, the meta-DSL implementation can implement behaviors that would otherwise need to be provided in the host. Thus a relatively simple set of host-language extensions can add a great deal of power.

While our prototype is a first step on the evolution path to bring Visser's vision to fruition in mainstream languages, he imagined a much wider set of capabilities. Much work remains to be done to adapt these ideas and we must adjust our approach for varying host languages. In particular, providing rich IDE services and type system integration both seem to require widening the interface for macros beyond simple syntax-to-syntax transformation.

References

1 Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax to textual code. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. `doi:10.1145/3428290`.

2 Michael Ballantyne, Alexis King, and Matthias Felleisen. Macros for domain-specific languages. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. `doi:10.1145/3428297`.

3 Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. Predictable macros for Hindley-Milner. In *The Workshop on Type-Driven Development*, TyDe '20, 2020. URL: `https://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf`.

4 Hendrik Bünder and Herbert Kuchen. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In *Model-Driven Engineering and Software Development*, MODELSWARD 2019, pages 225–245, 2020. `doi:10.1007/978-3-030-37873-8_10`.

5 Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent type systems as macros. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. `doi:10.1145/3371071`.

6 Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Proc. Principles of Programming Languages*, POPL 2017, pages 694–705, 2017. `doi:10.1145/3009837.3009886`.

7 Microsoft Corporation. Language server protocol. URL: `https://microsoft.github.io/language-server-protocol/`.

8 Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4-5):439–476, 2012. `doi:10.1017/S0956796812000275`.

9 Joe Darcy and Oracle. Java specification request 269: pluggable annotation processing. URL: `https://jcp.org/en/jsr/detail?id=269`.

10 Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Proc. Generative Programming and Component Engineering*, GPCE '11, pages 167–176, 2011. `doi:10.1145/2047862.2047891`.

11 Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based syntactic language extensibility. In *Proc. Object-Oriented Programming Systems, Languages & Applications*, OOPSLA '11, pages 391–406, 2011. `doi:10.1145/2048066.2048099`.

12 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, February 2018. `doi:10.1145/3127323`.

13 Daniel Feltey, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Languages the Racket way. *Language Workbench Challenge*, 2016. URL: `https://users.cs.northwestern.edu/~robby/pubs/papers/lwc2016-ffkscfff.pdf`.

14 Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. International Conference on Functional Programming*, ICFP '02, pages 72–83, 2002. `doi:10.1145/581478.581486`.

15 Matthew Flatt. Binding as sets of scopes. In *Proc. Principles of Programming Languages*, POPL '16, pages 705–717, 2016. `doi:10.1145/2837614.2837620`.

16 Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming*, 22(2):181–216, March 2012. `doi:10.1017/S0956796812000093`.

17 Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings. In *Proc. International Conference on Functional Programming*, ICFP '14, pages 339–347, 2014. `doi:10.1145/2628136.2628138`.

18 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proc. Object-Oriented Programming Systems, Languages & Applications*, OOPSLA '10, pages 444–463, 2010. `doi:10.1145/1869459.1869497`.

19 Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live GUIs. In *Proc. Programming Language Design and Implementation*, PLDI '21, pages 511–525, 2021. doi:10.1145/3453483.3454059.

20 Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. doi:10.1145/3527329.

21 The Rust Project. The Rust reference: Procedural macros. URL: https://doc.rust-lang.org/reference/procedural-macros.html.

22 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. Programming Language Design and Implementation*, PLDI '11, pages 132–141, 2011. doi:10.1145/1993498.1993514.

23 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

24 Eelco Visser. WebDSL: a case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School*, GTTSE '07, pages 291–373, 2007. doi:10.1007/978-3-540-88643-3_7.

25 Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. A language designer's workbench: a one-stop-shop for implementation and verification of language designs. In *Proc. International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 95–111, 2014. doi:10.1145/2661136.2661149.

26 Markus Voelter and Sascha Lisson. Supporting diverse notations in MPS' projectional editor. In *Proc. International Workshop on The Globalization of Modeling Languages*, GEMOC 2014, pages 7–16, 2014. URL: http://ceur-ws.org/Vol-1236/paper-03.pdf.

27 Martin P Ward. Language-oriented programming. *Software Concepts and Tools*, 15(4):147–161, 1994. doi:10.1007/978-1-4302-2390-0-12.

The Importance of Being Eelco

Andrew P. Black ✉ 🏠
Portland State University, OR, USA

Kim B. Bruce ✉ 🏠
Pomona College, Claremont, CA, USA

James Noble ✉ 🏠
Creative Research & Programing, Wellington, NZ

Abstract

Programming language designers and implementers are taught that:

semantics are more worthwhile than syntax,
that programs exist to embody proofs, rather than to get work done, and
to value Dijkstra more than Van Wijngaarden.

Eelco Visser believed that, while there is value in the items on the left, there is at least as much value in the items on the right. This short paper explores how Eelco Visser embodied these values, and how he encouraged our work on the Grace programming language, supported that work withio Spoofax, and provided a venue for discussion within the WG2.16 Programming Language Design working group.

2012 ACM Subject Classification Software and its engineering → Syntax; Software and its engineering → Semantics; Software and its engineering → Object oriented languages; Software and its engineering → Translator writing systems and compiler generators

Keywords and phrases Eelco Visser, Grace, Spoofax, syntax

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.4

Funding *James Noble*: This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden, Amazon Web Services, and Agoric.

1 Introduction

The year 2010 seems as if it was a long, long time ago, if not in a galaxy far, far away. Barack Obama had been president for a couple of years, Boris Johnson was up to only his second wife, and the idea of Donald Trump as a political figure was far from the mind of the most fevered cartoonist [42].

2010 will also be remembered for the year that OOPSLA was held in a casino in Reno, Nevada. Location aside, the meeting was an important one for the *dramatis personae* of this narrative. The authors of the present paper organized a panel session on the next educational programming language [3], which marked the beginning of our work on Grace. Lennart Kats and Eelco Visser received the Best Student Paper award for a summative article describing Spoofax [30] – hardly the beginning of that work, but a significant landmark; that paper was to receive an OOPSLA Most Influential Paper Award in 2020. Finally, in a restaurant somewhere under a freeway, the first organizational meeting took place of what would eventually become the IFIP Working Group on Language Design, which included Eelco, Andrew, James and (probably!) Kim [21]. Out of that organizational meeting grew plans for the first technical meeting, hosted by Jan-Willem Maessen and Mark Miller in Mountain View the following June.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 4; pp. 4:1–4:15
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we trace the way that the threads of Spoofax, Grace, and the Working Group on Language Design became woven together over the following twelve years, until Eelco's untimely death in April of 2022. We also explore some of what we believe are core values for language design and implementation: that syntax matters, that programs exist for communication and to be executed, and that the way we treat and are treated by our colleagues makes all the difference.

1.1 What is Spoofax?

The name Spoofax is the result of an engineering exercise conducted by Karl Trygve Kalleberg: "Spoofax" had zero hits in Google, and a free domain name! So wrote Eelco himself in a blog post about the history of Spoofax [49]. Spoofax was a response to the rise of the IDE, which extended the task of a language implementor from simply providing a compiler or interpreter, to also providing a range of editor services, such as syntax highlighting, code completion, outline views, and code formatting. In theory, IDEs such as Eclipse were extensible: all one needed to do was to write the right plugin. In practice, writing a plugin was far from simple. Spoofax *generated* the plugins, starting from declarative specifications of the syntax and semantics of the target language. Spoofax integrated a variety of tools: SDF2 [47] for specifying grammars, generators of customizable editor services based on the syntax, and (initially) the Stratego program transformation language [8] to describe semantics using rewrite rules. Spoofax evolved to encompass additional tools, in particular the DynSem [46] language for specifying dynamic semantics. Behind all of these tools was Eelco's drive to bring the power of modern computers to the task of language implementation.

It's not our goal here to describe Spoofax in detail. Those interested in learning more should start with the above-mentioned blog post [49], and follow up with the double-award-winning paper from 2010 [30].

1.2 What is Grace?

Grace is a programming language intended for the teaching of programming to students beginning the study of computer science. Grace is named after Rear Admiral Grace Murray Hopper, and in the hope that programs written therein would be *Graceful*.

In 2010, the programming languages landscape appeared moribund – especially for academics who were teaching introductory programming courses. According to the TIOBE index [41], Java, C, and C++ were the three most popular languages between 2007 and 2017. Python was slowly bubbling up (from number 8 in 2007 to number 5 in 2017), and the PLT Scheme project was just starting the process of changing the name of their language to Racket [1].

As teachers and academic researchers oriented towards objects, we had to choose a language for our teaching and research [17, 22, 31, 39, 50]. The general dissatisfaction with the options available for teaching came to a head at ECOOP 2010 in Maribor: since we were language designers and implementors, why not work on an object-oriented language targeted at our own needs? Haskell and Algol had been built by teams of academics; why couldn't we design and implement a new object-oriented language for teaching and research? After an initial flurry of interest sparked by the panel-discussion at SPLASH 2010 [38], the task of designing Grace was taken on by Black, Bruce and Noble, the authors of the present paper. The resulting language, as it stood around the time relevant to this article, is described in the short papers presented at SIGSCE [9] and IEEE CSEE&T in 2013 [37].

1.3 What is WG2.16?

Design of all kinds requires feedback. Architecture students learn this early on: they propose and defend their design ideas in charrettes. But computer scientists don't have that tradition: instead we have "publication venues", where we are expected to present our designs as if they were finished, rather than as sketches to invite feedback.

In July 2010, Jonathan Edwards and Andrew Black were discussing this problem in Portland. We roped in Gilad Bracha, and drafted a manifesto:

> A few of us feel that there is currently no good forum for discussing programming language design issues; researchers interested in language design are isolated and lack a place to exchange ideas and criticism. Computer science conferences no longer serve this role, because they have become fixated on rigorous evaluation. There is nothing wrong with rigorous evaluation of things that can be so evaluated, but language design ideas, particularly in their formative stages where feedback is most crucial – and when there may be neither implementation nor experience with their use – cannot be so evaluated. Indeed, many landmark papers on language design could not be published today: Liskov on data abstraction, or Parnas on modules, or Dijkstra on goto. These papers consisted of reasoned arguments for the power of a new idea. We need a venue where such ideas and arguments are still welcome and where they can be refined by constructive criticism.

A number of our colleagues signed on to this manifesto, which eventually because a proposal to IFIP Technical Committee 2 (Software: Theory and Practice) for the creation of a new working group on programming language design. At the zeroth meeting in Mountain View in 2011, Eelco Visser was elected as chair of this loose group. Eeclo took the proposal to TC2, gained their approval, and the group become IFIP WG2.16 – the Working group on Programming Language Design

2 A Little Grace

Our interactions with Eelco were primarily in the context of the Grace programming language design project. While paper is not a primer on Grace, some discussion of two of Grace's peculiarities is necessary to provide background for the story of our interactions with Eelco and the Spoofax team.

2.1 Objects and Classes

A Grace object is created by an object constructor, a special kind of expression introduced by the reserved word **object**. A Grace **class** is a declaration that names and parameterises an object constructor. Here is an example:

```
object {
    def name is public = "Fido"
    var age is public := 0
    method say(phrase : String) {
        print "{name} says {phrase}"
    }
    print "{name} has been born."
}
```

```
class dog(n:String) {
    def name is public = n
    var age is public := 0
    method say(phrase : String) {
        print "{name} says {phrase}"
    }
    print "{name} has been born."
}
dog "Fido"
```

The code on the left is an expression that creates a new object. On the right is a class declaration; it declares a *method* named dog(_) whose body is a parameterized version of the object constructor on the left. The request dog "Fido" requests execution of this method, and thus creates a new object. Both fragments print “Fido has been born.”

2.2 Syntax and Layout

Because Grace was intended for teaching, its syntax was designed to be writable by novices and readable by anyone familiar with another object-based language. We chose a relatively conventional mix of the “curly bracket” style of C and the keyword style of Pascal. Like C, declarations and code blocks are delimited by {. . .} rather than begin. . . end, but like Pascal, all declarations are marked by keywords (e.g., **def**, **var**, **method**). We hoped that this would make the syntax clearer to novices, as well as teaching them important vocabulary. Newlines terminate statements, making semicolons optional, and eliminating a common source of frustration. Control structures are not built in; instead they are methods that use Grace’s multi-part method names, inspired by method names in Smalltalk but modified to eliminate the colons. As a consequence, keywords appear between the components: “if (flag) then { . . . }” rather than “if (flag) { . . . }”.

We debated long and hard whether Grace should be a “curly bracket language” like C or an “indentation-sensitive” language like Python. We ended up requiring both! We chose braces to delimit blocks, because Grace blocks are λ-expressions, and we wanted a compact in-line form for simple function such as $\lambda x.x + 1$, which is written in Grace as {x → x + 1}. The story in Grace is that “braces are suspenders”: code enclosed by braces is not executed immediately, but is stored away so that it can be executed later (i.e., is represented as a closure). This is consistent with the use of braces to enclose method bodies.

As well as using braces to indicate the boundaries of code blocks and declarations, Grace requires that code layout must be consistent with these boundaries. That is, indentation must increase after an opening brace, and return to the prior level with (or after) the matching closing brace. Here is a short example:

```
1  def direction = if (a > b) then {
2      "up"
3  } elseif { a < b } then {
4      "down"
5  } else {
6      "fixed"
7  }
8  while {stream.hasNext} do {
9       print(stream.read)
10 }
```

The expression on the right-hand side of the **def** on line 1 uses parentheses to enclose the condition (a > b), because that condition is always evaluated, and indeed is evaluated before the if(_)then(_)elseif(_)then(_) is executed. However, the blocks after then, elseif, and else must be evaluated only when necessary: it is to make this possible that they are wrapped in braces. The while(_)do(_) statement is similar: because it requires repeated evaluation of both the while condition and the body of the do, they must both be wrapped in braces.

While indentation must correspond to the brace structure, it is also used to determine the span of statements, and thus the elision of semicolons. In particular, indentation distinguishes between a single method request with a multi-part name, and multiple requests with single part names. The left and centre layouts below will both work for Grace's if(_)then(_)else(_) control structure: the one on the left indents the blocks following then and else, while the center example used indentation *not* following an open brace to turn the three physical lines into a single logical line. In contrast, the rightmost layout will be interpreted as three separate method requests, on three separate lines: an if(_), a then(_), and an else(_):

```
if (condition) then {
    doThis
} else {
    doThat
}
```

```
if (condition)
    then { doThis }
    else { doThat }
    //one request
```

```
if (condition)
then { doThis }
else { doThat }
// three separate requests
```

2.3 Nesting and Inheritance

Grace objects and classes can inherit from other classes via **inherit** (and the closely related **use**) clauses. For example: we can define an object out that inherits from a superClass:

```
class superClass {
    method m { "in superclass." }
}
def out = object {
    inherit superClass
    method foo { print (m) }
}
out.foo
```

The program above will print "in superclass." Note that Grace follows C++, in allowing the programmer to elide **self** in method requests, and Eiffel and Self with a single syntax and namespace for both methods and fields. So a request of m (meaning **self**.m) in the body of method foo is an example: here that resolves to the foo method declaration in superClass: but the same request could equally resolve to a constant definition, to reading a variable, or even instantiating a class.

In addition, Grace's objects – like those in most contemporary object-oriented languages following BETA [32] – can be lexically nested:

```
def out = object {
    method m { "in enclosing object." }
    def inner is public = object {
        method foo { print (m) }
    }
}
out.inner.foo
```

The program above will print "in enclosing object." Note the method foo is lexically inside *two* objects: the object inner and the object out. What then is the meaning of the unqualified m in print(m)? We can see that it cannot mean **self**.m because **self** – the object inner – does not have an m. We deduce that it must mean **outer**.m, that is, the m defined in the object lexically surrounding **self**. The devil is always in the details, or as an earlier draft of this paper said instead: "the ogre is in the orthogonality". Although comically alliterative, that remark is inaccurate: it is the sort of remark that gives orthogonality an unjustifiably bad reputation (see Section 4.3). A moment's thought should make us realize that nesting and implicit **self** **cannot** be orthogonal: implicit **self** is dubious [34] in a context in which there are multiple current objects.

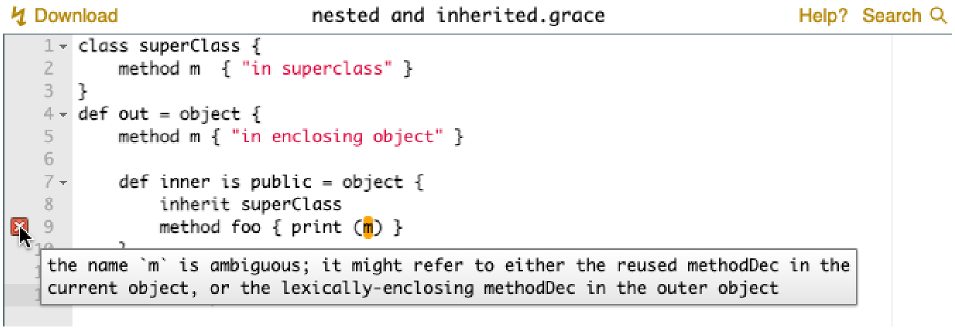
What if a program attempts to use nesting, inheritance and implicit **self** at the same time, as in the example below?

```
class superClass {
    method m { "in superclass." }
}
def out = object {
    method m { "in enclosing object." }

    def inner is public = object {
        inherit superClass
        method foo { print (m) }
    }
}
out.inner.foo
```

Which m does the method foo invoke: the m in the lexically-enclosing object out, or the m inherited from superclass? This is exactly the sort of question that a language workbench such as Spoofax should help one explore.

This potential ambiguity is common across many object-oriented languages – with as many different solutions as there are languages [7]. Java uses "up then out" semantics, and thus would invoke m inherited from the superclass. Newspeak uses "out then up", so would invoke m in the enclosing object. As a language designed for education, Grace bans such ambiguous requests, requiring that the programmer resolve the ambiguity by writing **self**.m or **outer**.m, as shown in Figure 1 [6, §implicit-requests]. Nonetheless, the fact that the question "Which m?" can be asked at all means that a complete specification of the language must answer it.



```
class superClass {
    method m  { "in superclass" }
}
def out = object {
    method m { "in enclosing object" }

    def inner is public = object {
        inherit superClass
        method foo { print (m) }
```

the name `m` is ambiguous; it might refer to either the reused methodDec in the current object, or the lexically-enclosing methodDec in the outer object

Figure 1 Grace IDE showing ambiguous implicit request.

3 Grace in Spoofax

The original goal of the Grace project was to produce a language specification, not a language implementation [4, 5, 10]. While at least one implementation would be essential to guide the process of writing the specification, we hewed to the 20th century ideal that a programming language should be implementation independent. Build it (the specification), we thought, and they (the implementors) will come. How naïve we were![1]

[1] Not just naïve, but historically wrong: pretty much every successful programming language since SIMULA has been based on a single canonical implementation.

3.1 SDF2 Grace Parser

But come they did, or rather, Eelco Visser and his Spoofax team came: notably master's student Michiel Haisma and doctoral students Vlad Vergu and Luis Eduardo de Souza Amorim. We recall Eelco attending, slightly bemused, the meeting at ECOOP 2010 where the Grace project was mooted. He was too wise to sign on to the SPLASH 2010 "Manifesto" [5], but as one of the forces behind what became IFIP WG2.16, he was part of the audience for the Grace design effort. By the time of the first official meeting of WG2.16 (in London, in February–March 2012), the first iteration of Grace's design was complete. In an email exchange between Eelco and James Noble, following up on a "conversation after the pub", Eelco expressed interest in becoming an early implementor of Grace. We were of course delighted: as Eelco's involvement promised incisive design feedback and a soild implementation – while Eelco would benefit from practical experience with Spoofax on a language not of his own design. Indeed, Eelco went as far as to say:

> Rather than farming this out to a student, I'm planning to make it my 'trying out new features of Spoofax and learning about design choices in (OO) language design' project, with all risks associated with that, so don't hold your breath.[2]

Eelco was as good as his word. Working off an early grammar for Grace (at the time, represented using a parser combinator library within Grace itself) by SPLASH in October 2012 Eelco had the bones of a Spoofax parser working for Grace. Somehow, he had written this in his spare time! The Spoofax parser could handle essentially all the language as defined at the time, with the exception of Grace's then ill-defined layout rules: rather than relying on indentation and line breaks, Spoofax-Grace statements had to be terminated with semicolons, and there was no enforcement of Grace's requirement that indentation be consistent with brace-structure.

3.2 Spoofax–Grace

Eelco's parser was then extended by Michiel Haisma for his Master's thesis, resulting in a fairly complete implementation of the core of Grace completely within the Spoofax environment. (One of our initial goals for the Grace project was that the language should be implementable by a couple of graduate students in about a year: Haisma's thesis demonstrates that this goal could be met by talented students using the right tools).

Up to this point, Grace's specification was informal, and existing implementations were hand-coded interpreters and compilers. The aim of Spoofax–Grace was not just to provide an implementation of the Grace programming language, but also to serve as a reference implementation that could be tested, and as a specification that could be easily read, understood and changed [23, 24, 45].

Figure 2 shows the architecture of Spoofax–Grace. Spoofax's SDF3 DSL [15] parses Grace code into an initial AST. Next, the Stratego transformation language rewrites some Grace constructs (such as classes and traits) that are defined in terms of simpler constructs (such as methods and objects) in a "desuguring" pass. A "lowering" pass then produces a canonical, fully decorated AST [8]. Finally, definitions in the DynSem DSL [46] are used to interpret the program represented by the lowered AST.

[2] Email message from Eelco Visser to James Noble dated 4 March 2012.

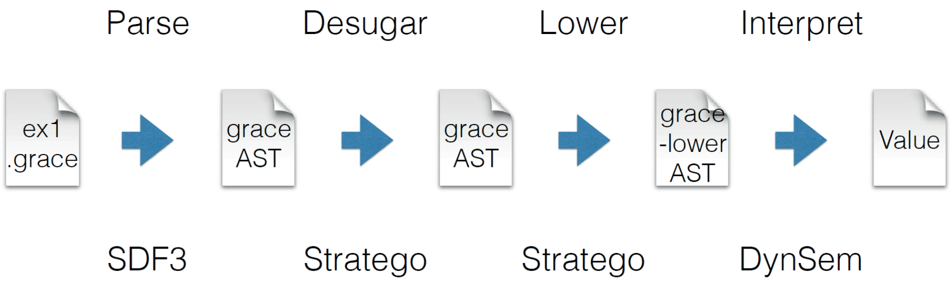


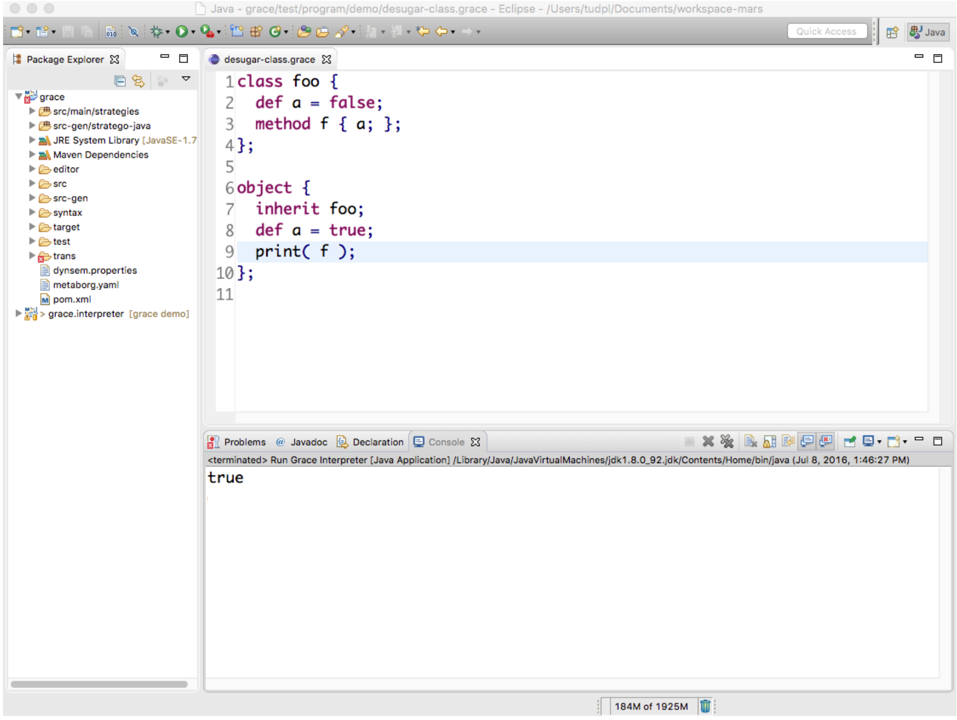
Figure 2 Spoofax–Grace Architecture. From Michiel Haisma, "Grace in Spoofax" [23], used with permission.

Figure 3 shows the system running a simple Grace program inside Eclipse. The leftmost column contains the Eclipse explorer; the central window shows the Grace source code – pretty-printed and syntax-coloured automatically from the Spoofax definitions. The bottom window shows the output of the DynSem interpreter executing the Grace program – here, simply: `true`.

The Spoofax–Grace project illuminated some details of the Grace specification as it existed at the time, and made us realize that the specification was not as precise as we had thought. One important area was the semantics of name resolution: the part of the language that had to combine the local definitions within an object with the definitions inherited from superclasses, reused from traits, and located in enclosing lexical scopes, which include the the outermost scope that defines the module's dialect [27, 36]. Based upon Eelco's theory of Scope Graphs [35, 43], Grace's name resolution and request lookup semantics were encoded as part of the overall operational semantics using the Spoofax DynSem DSL [46]. The DynSem implementation was shorter and easier to modify than the existing Grace implementations [45], and also clarified that the computational complexity of a Grace method request in an object n nested levels deep, with p parent objects (traits and superclasses), was $O(np)$[3].

The Spoofax implementation of Grace was actually more general than we, as the designers of Grace, had ever intended. Because we had always planned for Grace to have a static type system, it was important throughout our design discussions that the *shape* of a Grace object – the methods and fields that it contained – could be determined statically. Although **inherit** and **use** statements describe the parent object (the object being reused) with *expressions*, we intended that these expressions be *manifest*, that is, evaluable statically. However, the early specification documents didn't make this sufficiently clear, and Vergu *et al.* write: "The use of expressions to determine ancestors means that meaningful name resolution can only be performed at run time" [45]. The Spoofax implementation agreed with our prototype implementation in the sense that any Grace program that was accepted by the prototype gave the same results in Spoofax, but the Spoofax–Grace implementation allowed for reuse of objects whose shapes could not be ascertained until run time. This experience was enlightening to all involved, and made clear that we needed to do some serious work on our language specification – in particular, to reconsider the definition of *manifest*.

[3] To see why, consider an object n levels deep, where each enclosing defintion inherits from p classes or traits. This means there are $O(np)$ possible positions where m could potentially be declared – basically once in every lexically enclosing definition, and once in every definition inherited into each lexical scope. Grace's rules about ambiguity and (lack of) overloading means that every potential position must be checked, not just the positions that are inhabited.



```
class foo {
  def a = false;
  method f { a; };
};

object {
  inherit foo;
  def a = true;
  print( f );
};
```

```
<terminated> Run Grace Interpreter [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Jul 8, 2016, 1:46:27 PM)
true
```

Figure 3 Grace running in Eclipse, using plugins generated by Spoofax. From Michiel Haisma "Grace in Spoofax" [23], used with permission.

The purpose of the Spoofax–Grace project was as much to evaluate the Spoofax toolset as the Grace specification. Other than the difficulty of handling layout, the toolset performed admirably: deficiencies of Grace-Spoofax (missing pattern matching, lack of a type system and static analyses) were due more to a lack of time, or to imprecision in the language specification, than to weaknesses in the tools. The Spoofax implementation of Grace is still available [25], although not maintained, and now includes a parser that can handle Grace's layout, based on extensions to SDF3 that were made while the main Spoofax–Grace project was coming to an end [15, 16].

Looking back over this collaboration, it's clear the result was successful for both parties. Eelco's team obtained valuable practical experience, which eventually led to extensions to SDF for layout parsing; while Grace gained significant feedback on both the langauge design and its presentation in the language specification document. One important lesson was that a clear separation between static and dynamic semantics would have been beneficial to both language designers and to Spoofax users. There are places where the Grace specification deliberately leaves open the question of whether a check is static or dynamic, to allow the implementor more freedom. However, when intended, this should be done explicitly, e.g.: "The checks necessary to implement this guarantee [type safety] may be performed statically or dynamically". Another lesson – in a quite different dimension – is the value of the Agile practice of the on-site customer [2, p. 60]: if the Grace and Spoofax teams had been co-located, the lack of clarity about what could be inherited would have been discovered much sooner.

4 Being Eelco

This paper makes some presumptuous claims about the core values of language design and implementation, and the way in which Eelco Visser embodied those values. If Eelco were still with us, we could do this cavalierly, knowing that Eelco would take our comments in good heart, and enjoy disputing with us. Sadly, that will not happen, so we proceed with more caution. We will do our best to justify these claims, and leave it to the reader to decide if they have value.

4.1 Semantics vs. Syntax

We are going to say it outright: syntax is important! We do not claim that semantics are *unimportant*, but that syntax must come first: the semantics has to be attached to something. Syntax *carries* the semantics: the vast majority of programmers will only approach semantics via syntax [12, 14]. A well-designed syntax should suggest the appropriate semantics; if a naïve reading of the syntax is at variance with the semantics, then one or the other needs to change.

In Spoofax, Eelco acknowledged that syntax matters to programmers. Parsing, pretty-printing, and editor support are important. They are, or ought to be, the cornerstone of any language implementation. It is certainly possible to produce a language workbench that ignores syntax – the input language could be S-expressions – and focuses instead on semantics, optimizations, and execution. But that would set aside a lot of what legitimately concerns users, and abdicates responsibility to help implementors in an area where tooling is both important and effective.

4.2 Program Proofs vs. Working Code

Looking through the proceedings of computer science conferences, where one used to find descriptions of working programming *systems*, one now finds descriptions of formal calculi – Featherweight Java, System F, and so on. One can see traces of this trend as far back as 1979, when Dijkstra thought it appropriate to ridicule Teitelman's Interlisp system because the "reference manual for Interlisp is already something like a two-inch thick telephone directory" [20]. Having an extensive library was apparently a fatal flaw in Dijkstra's eye[4].

Yes, there is value in formal calculi, and there is value in proofs of correctness. There is also value in complexity theory and in choosing an appropriate algorithm. But the *reason* that these things have value is because programs do stuff in the real world, we want them to do the *right* stuff, and we want it done quickly.

Eelco, as exemplified in Spoofax, was interested in a system that worked in the real world – for example, that integrated with Eclipse, and provided programmers with editing tools that were satisfying to use. Yes, the Spoofax tools were built on sound theoretical foundations. But foundations alone were not enough: Spoofax had to get work done.

As language designers, we appreciate the value of formal systems. When one changes one's grammar, it's nice to know that it will still parse all of one's test cases. When one changes one's type system, it's nice to know that the type system remains sound. But there is also enormous value in having a working implementation on which you can run examples. We can remember occasions where, after long discussions and some longer walks, we agreed

[4] "For the absence of a bibliography I offer neither explanation nor apology."
Preface to Dijkstra's *A Discipline of Programming* [18].

on a change to Grace. Then one of us started programming in the revised language, and was forced to confront the consequences of the change! Of course, if we had only been smarter, we could perhaps have foreseen these consequences. Alas, we are who we are. Having an implementation that could quickly show us the consequences of a change, and show it on a sizable body of code, was of enormous value during the design process.

We wish that we could write: "And such an implementation is just what Eelco and Spoofax had provided." Unfortunately, the timing did not work out. Because the Spoofax implementation did not originally recognize Grace's indentation rules, and all of our existing code used indentation rather than semicolons, we could not run our code on Spoofax-Grace. Our design was indeed guided by an implementation, but not one created in Spoofax. Instead we used a self-hosting compiler written in Grace itself, based on an early implementation written by Michael Homer, who at the time was a student at Victoria University of Wellington [26]. A self-hosting compiler had some advantages – it gave us experience with a substantial body of real Grace code – but did not provide a readable syntax or semantics. Nevertheless, we derived great benefit from the Spoofax implementation. Designing a language can be a lonely business; the fact that another team in another country were *interested* in what we were doing, to the point of taking our specification and implementing it, was heartening and rewarding.

4.3 Dijkstra vs. Van Wijngaarden

Although Adriaan van Wijngaarden was Edsger Dijkstra's boss and doctoral supervisor, the two men could hardly have been more different. Let us concentrate here on just one difference: where Van Wijngaarden was an enthusiastic adopter of new technology, Dijkstra seems to have been less interested in what he called "gadget development"[20].

This may appear to be an odd comment to make about one of the pioneers of our science, but there is evidence aplenty. Dijkstra made original contributions to the design of programming calculi and to the axiomatic method for reasoning about programs. Dijkstra, however, seemed unwilling to accept that working to improve programming technology beyond the imperative languages where he made his mark was a worthwhile activity, not only for himself, but for anyone else! His dismissive review of John Backus' Turning Award lecture [19] is a case in point; those interested in exploring this particular issue further should read the archive of the subsequent correspondence between Backus and Dijkstra [13]. Dijkstra's point of view seemed to be that if only everyone were smarter, or thought more, or had more mathematical training, then the deficiencies of our science could be overcome. New technology was not required, and would not help: what was required was a new generation of better trained practitioners.

Van Wijngaarden was from a different mould. He aimed to contribute to a group, rather than work as as solitary individual. He enthusiastically adopted new technology where it would solve a recognized problem, and was ready to try out new ideas. He was troubled by the inability of BNF (developed for the definition of Algol 60) to express context conditions; for the definition of Algol 68 he developed a new technology, the two-level grammar, that overcame this deficiency [44].

In the years since their invention, two-level grammars, also known as W-grammars after Van Wijngaarden, have acquired a bad reputation. The notion of "orthogonality" in language design, much facilitated by the use of a W-grammar for language description, has faired little better. Perhaps this is because the adoption of W-grammars by WG2.1 for the description of what became Algol 68 can be seen as the beginning of the schism that led to the Algol 68 minority report and significant resignations from WG2.1. Nevertheless, it's worthwhile to look at the *ideals* carried by W-grammars and Van Wijngaarden's notes on "*Orthogonal design*" (MR76) [44]:

1. That all the data values that can be manipulated in the programming language should be "first class". In other words, if you can put one type of value (such as a number) into a list, or pass it as an argument, or return it as a result, then you should be able to do the same with other types of values (such as strings, records, and functions).
2. That the effect of declarations should be recorded in a syntactic structure, along with their types, and later references to declared names should be looked up in this structure to ascertain whether or not they have been declared, and if so, to find their types.
3. That languages can be *expression-oriented*, in that every construct can be considered as an expression whose execution delivers a value of some type (as well as having a possible effect on the store), rather than distinguishing between statements and expressions.

Independent of whether we like them or not, all of these contributions have won in the marketplace of ideas. Indeed, purely syntactic approaches to type-checking [28, 52] (Item 2 above) have essentially displaced all others. Two-level grammars also gave Van Wijngaarden a mechanism for uniformly generating productions for *sequences* of entities, *parenthesized* entities, and so on, without inventing an unnecessary diversity of special-purpose notations [51]. Our point is that faced with a need, Aad van Wijngaarden was willing to use, or invent, technology to address it.

What then is the problem with W-grammars? Primarily, that they are *too* powerful. As early as 1967, Michael Sintzoff [40] showed[5], that any recursively-enumerable set can be described by a W-grammar. This means that the parsing problem for W-grammars is *in general* undecidable. Of course, when using a W-grammar to describe a programming language, the author *intends* that the language be parsable, and there have been many attempts to place formal restrictions on W-grammars to ensure that parsing is possible. Early in his career (1998), Eelco authored one such attempt [48], which also makes useful connections between W-grammars and algebras, and between orthogonality and polymorphism.

Another instance of Van Wijngaarden's willingness to embrace technology, particularly appropriate as a contrast to Dijkstra's preference for writing with a fountain pen, is Van Wijngaarden's use of the IBM Selectric typewriter. In "A History of Algol 68", Charles Lindsey writes [33]:

> The use of a distinctive font to distinguish the syntax (in addition to italic for program fragments) commenced with [MR 93], using an IBM golf-ball typewriter with much interchanging of golf balls. Each time Van Wijngaarden acquired a new golf ball, he would find some place in the Report where it could be used (spot the APL golf ball in the representations chapter). In fact, he did much of this typing himself (including the whole of [MR 101]).

Van Wijngaarden may have done the typing himself because of the inability of the typists at the Mathematisch Centrum to distinguish a roman period "." from an italic period "*.*" [11]. The point is that Van Wijngaarden was willing to do relatively mundane work himself, adopting the latest technologuy to do so.

Eelco Visser was a man very much in the Van Wijngaarden mould – in attitudes and interactions, if not as nattily dressed. (As far as we can ascertain, they are not related academically, other than both being Dutch.) He was willing and able to harness technology to get things done: Eclipse is, after all, merely the 2010s version of a golf-ball typewriter [29]. He created institutions – his research group at Delft WG2.16, – that reified those values. He

[5] We should write: "is reputed to have shown" because we haven't actually found a copy. However, Eelco references this work in his 1998 paper [48], so perhaps that's good enough?

built tools, not just Spoofax, but also WebDSL, and the researchr conference system, that meant other people could get their own work done. He genuinely cared for those around him, be they students or colleagues.

5 Conclusion

Eelco Visser was a kind man and a sympathetic colleague. His many accomplishments never went to his head. Instead of belittling those who did not or could not follow, he gave them a helping hand. Eelco was endlessly patient as the Grace authors tried to come to grips with Spoofax.

Andrew Black treasures happy memories of a visit to Delft, arranged by Eelco as a means to pay for a trip to SPLASH in Amsterdam. The visit was rich with interactions with the members of his group, after which we and some students rode our bikes around Delft *en route* to dinner. Eelco contributed in many other ways to the success of our profession, in particular by supporting SIGPLAN conferences with the conf.researchr.org website, and of course by helping to create and chair IFIP WG2.16. Eelco is sorely missed.

References

1 Eli Barzilay. Racket, June 2010. URL: `https://blog.racket-lang.org/2010/06/racket.html`.

2 Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition, 1999.

3 Andrew Black, Kim B. Bruce, and James Noble. Designing the next educational programming language. In *Proceedings ACM international conference on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 201–204, New York, NY, USA, October 2010. ACM. `doi:10.1145/1869542.1869574`.

4 Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*, pages 85–98, New York, NY, 2012. ACM. `doi:10.1145/2384592.2384601`.

5 Andrew P. Black, Kim B. Bruce, and James Noble. Panel: designing the next educational programming language. In *SPLASH/OOPSLA Companion*, 2010.

6 Andrew P. Black, Kim B. Bruce, and James Noble. Grace language specification, version 0.8.2. `http://web.cecs.pdx.edu/~grace/doc/lang-spec/`, July 2020.

7 Gilad Bracha. On the interaction of method lookup and scope with inheritance and nesting. In *3rd ECOOP Workshop on Dynamic Languages and Applications*, 2007.

8 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17: A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

9 Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *Proceedings 44th SIGCSE Technical Symposium on Computer Science Education*, pages 129–134. ACM, 2013. `doi:10.1145/2445196.2445240`.

10 Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*, 2013.

11 Centrum Wiskunde & Informatica. Memories of Aad van Wijngaarden (1916-1987), November 2016. URL: `https://www.youtube.com/watch?v=okLiv1QA4Dg`.

12 Daniel Chandler. *Semiotics: The Basics*. Routledge, 2002.

13 Jiahao Chen. “This guy’s arrogance takes your breath away”: Letters between John W Backus and Edsger W Dijkstra, 1979. Blog entry, May 2016. URL: https://medium.com/@acidflask/this-guys-arrogance-takes-your-breath-away-5b903624ca5f [cited 20 Nov 2022].

14 Paul Cobley and Litza Jansz. *Semiotics for Beginners.* Icon Books, Cambridge, England, 1997.

15 Luís Eduardo de Souza Amorilm and Eelco Visser. Multi-purpose syntax definition with sdf3. In *Software Engineering and Formal Methods*, 2020.

16 Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages. In *SLE*, 2018.

17 Charles Dierbach. Python as a first programming language. *J. Comput. Sci. Coll.*, 29(6):153–154, June 2014.

18 E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

19 E.W. Dijkstra. A review of the 1977 Turing award lecture by John Backus (EWD692). Edsger W. Dijkstra Archive at Univ. Texas, undated, around November 1978. URL: https://www.cs.utexas.edu/users/EWD/ewd06xx/EWD692.PDF.

20 E.W. Dijkstra. Trip report E.W.Dijkstra, Mission Viejo, Santa Cruz, Austin, 29 July – 8 September 1979 (EWD714). Edsger W. Dijkstra Archive at Univ. Texas, September 1979. URL: https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD714.PDF.

21 Jonathan Edwards. Reno 2010 [online]. October 2010. URL: https://languagedesign.org/meetings/Reno2010.html [cited Jan 2023].

22 Diwaker Gupta. What is a good first programming language? *ACM Crossroads*, 10(4):7, August 2004.

23 Michiel Haisma. Grace in Spoofax. Master’s thesis, TUDelft, May 2017.

24 Michiel Haisma, Vlad Vergu, and Eelco Visser. Grace in Spoofax: Readable specification and implementation in one. In *Grace workshop at ECOOP*, July 2016. URL: https://2016.ecoop.org/details/GRACE-2016/2/Grace-in-Spoofax-Readable-Specification-and-Implementation-in-One.

25 Michiel Haisma, Vlad Vergu, and Eelco Visser. Spoofax-based implementation of the Grace language, February 2017. URL: https://github.com/MetaBorgCube/metaborg-grace.

26 Michael Homer. *Graceful Language Features and Interfaces.* PhD thesis, Victoria University of Wellington, 2014.

27 Michael Homer, Timothy Jones, James Noble, Kim B Bruce, and Andrew P Black. Graceful dialects. In Richard Jones, editor, *ECOOP*, volume 8586 of *LNCS*, pages 131–156. Springer, 2014.

28 Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

29 Poul-Henning Kamp. Sir, please step away from the ASR-33! to move forward with programming languages we need to break free from the tyranny of ASCII. *Queue*, 8(10):40–42, October 2010.

30 Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proc. ACM Int. Conf. Object Oriented Programming Systems Languages and Applications*, OOPSLA ’10, pages 444–463, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869497.

31 Laserfiche contributor. How your first programming language warps your brain [online]. Undated. URL: https://www.laserfiche.com/ecmblog/programming-languages-change-brain.

32 Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

33 C. H. Lindsey. A history of Algol 68. In *History of Programming Languages – II*, pages 27–96. Association for Computing Machinery, New York, NY, USA, 1996. doi:10.1145/234286.1057810.

34 Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP Proceedings*, 1999.

35 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *ESOP*, pages 205–231, 2015.

36 James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. Grace's inheritance. *Journal of Object Technology*, 16(2):2:1–35, April 2017.

37 James Noble, Michael Homer, Kim B. Bruce, and Andrew P. Black. Designing Grace: Can an introductory programming language support the teaching of software engineering. In *IEEE Conference on Software Engineering Education and Training (CSEE&T)*, 2013. URL: `http://gracelang.org/documents/cseet13main-id92-p-16403-preprint.pdf`.

38 Jack Rosenberger. Grace: A manifesto for a new educational object-oriented programming language. Blog@CACM, October 2010. Retrieved Jan 2023. `http://cacm.acm.org/blogs/blog-cacm/100389`.

39 Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. Language choice in introductory programming courses at Australasian and UK universities. In *Proc. 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 852–857, 2018.

40 Michel Sintzoff. Existence of a Van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société scientifique de Bruxelles, Série 2*, 81:115–118, 1967.

41 TIOBE Index for June 2022. `https://www.tiobe.com/tiobe-index`, 2022.

42 G.B. Trudeau. *Yuge!: 30 Years of Doonesbury on Trump*. Andrews McMeel, 2016.

43 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, 2016.

44 A. van Wijngaarden. *Orthogonal design and description of a formal language*. Rekenafdeling: MR. Stichting Mathematisch Centrum, 1965. MR 76. URL: `https://ir.cwi.nl/pub/9208/9208D.pdf`.

45 Vlad Vergu, Michiel Haisma, and Eelco Visser. The semantics of name resolution in Grace. In *DLS*, pages 63–74, 2017.

46 Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for dynamic semantics specification. In *26th Int. Conf. Rewriting Techniques and Applications (RTA '15)*, pages 365–378, June–July 2015.

47 Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Progr. Research Group, University of Amsterdam, July 1997. URL: `https://eelcovisser.org/publications/1995/Visser95.pdf`.

48 Eelco Visser. Polymorphic syntax definition. *Theoretical Computer Science*, 199(1–2):57–86, 1998. `doi:10.1016/S0304-3975(97)00268-5`.

49 Eelco Visser. A brief history of the Spoofax language workbench [online]. February 2021. URL: `https://eelcovisser.org/blog/2021/02/08/spoofax-mip/` [cited January 2023].

50 Richard L. Wexelblat. The consequences of one's first programming language. In *SIGSMALL*, pages 52–55, 1980. `doi:10.1145/800088.802823`.

51 Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *CACM*, 20(11):822–823, 1977.

52 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

Spoofax at Oracle: Domain-Specific Language Engineering for Large-Scale Graph Analytics

Houda Boukham ✉
Mohammed V University in Rabat,
Ecole Mohammadia d'Ingénieurs, Morocco
Oracle Labs, Casablanca, Morocco

Guido Wachsmuth ✉
Oracle Labs, Zürich, Switzerland

Toine Hartman ✉
Oracle Labs, Utrecht, The Netherlands

Hamza Boucherit ✉
Oracle Labs, Casablanca, Morocco

Oskar van Rest ✉
Oracle, Redwood Shores, CA, USA

Hassan Chafi ✉
Oracle Labs, Zürich, Switzerland

Sungpack Hong ✉
Oracle Labs, Redwood Shores, CA, USA

Martijn Dwars ✉
Oracle Labs, Zürich, Switzerland

Arnaud Delamare ✉
Oracle Labs, Zürich, Switzerland

Dalila Chiadmi ✉
Mohammed V University in Rabat,
Ecole Mohammadia d'Ingénieurs, Morocco

Abstract

For the last decade, teams at Oracle relied on the *Spoofax* language workbench to develop a family of domain-specific languages for graph analytics in research projects and in product development. In this paper, we analyze the requirements for integrating language processors into large-scale graph analytics toolkits and for the development of these language processors as part of a larger product development process. We discuss how *Spoofax* helps to meet these requirements and point out the need for future improvements.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Translator writing systems and compiler generators

Keywords and phrases language workbench, domain-specific language

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.5

Acknowledgements The following people contributed to Spoofax implementations of domain-specific languages for graph analytics as presented in this paper (in alphabetical order): Alexander Weld, Amin Ait Lamqadem, Arnaud Delamare, Calin Iorgulescu, Daco Harkes, Daniel Lehmann, Daniel Pelsmaeker, Danny Groenewegen, Davide Basilio Bartolini, Eelco Visser, Felix Kaser, Gabriel Konat, Guido Wachsmuth, Hamid El Maazouz, Hamza Boucher, Houda Boukham, Hugo Kapp, Ivo Wilms, Jan van der Lugt, Jeff Smits, Jinha Kim, Jinsoo Lee, Korbinian Schmid, Luis Eduardo de Souza Amorim, Manuel Then, Martijn Dwars, Martin Sevenich, Matthijs Bijman, Mohammed Bacha, Moritz Baumann, Mourad Abbay, Omar Heddi, Oskar van Rest, Ossama Elbourakkadi, Patrick Gruntz, Ragahavan Raman, Rob Vermaas, Siegfried Depner, Stefan Kaestle, Sungpack Hong, Toine Hartman, Vlad Vergu.

1 Introduction

In 2011, the *Parallel Graph AnalytiX* (*PGX*) team at Oracle Labs faced a challenge in its *Green-Marl* [1] compiler. The C++ implementation became increasingly difficult to maintain and slowed down the exploration of powerful optimization techniques. The *PGX* team was looking for alternatives, and identified declarative language specifications as advocated by the *Spoofax* project [2] as a promising approach to reduced maintenance efforts and to faster exploration of potential optimizations. In 2012, Oracle started a collaboration with Eelco

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 5; pp. 5:1–5:8
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Visser's research group at Delft University of Technology. Eelco's group started to develop a first language definition of *Green-Marl* in *Spoofax*, consisting of an *SDF2* [11, 12] grammar, name binding rules in the new name binding language *NaBL* [4], and type specifications in *TS* [13], an emerging meta-language for type specifications. Green-Marl provided a valuable industrial case study for the work on *NaBL* and *TS*, and appeared in several publications on *Spoofax* and its metalanguages. In 2013, the *PGX* team took over the work on the frontend and started to develop *Stratego* transformation rules for a compiler backend in *Spoofax*, targeting a Java-based runtime for large scale parallel graph analytics [8].

A decade of collaboration later, teams at Oracle rely on *Spoofax* to develop a family of domain-specific languages for graph analytics in research projects and in product development:

Green-Marl is a DSL for algorithmic graph processing [1]. We use it internally to implement over 60 graph algorithms which are available in Oracle products for graph analytics.

PGX Algorithm is another DSL for algorithmic graph processing [7]. It provides the same domain-specific constructs as *Green-Marl*, but captures them in a *Java*-API. *PGX Algorithm* is part of several Oracle products for graph analytics, allowing customers to implement their own graph algorithms.

PGQL is an *SQL*-like graph query language [6, 10] and is integrated into several Oracle products.

These languages and their language processors are building blocks of larger toolkits and Oracle products for graph analytics. In this paper, we analyze the requirements for integrating language processors into large scale graph analytics toolkits (Section 2) and for the development of these language processors as part of a larger product development process (Section 3). We discuss how *Spoofax* helps to meet these requirements and point out the need for future improvements.

2 Domain-Specific Language Processors for Graph Analytics

Large-scale graph analytics solutions typically adopt a client-server architecture. Users interact with a client such as a console or a notebook, while the actual graph data is processed on a server. Domain-specific languages such as Neo4j's *Cypher* query language [5], Apache TinkerPop's *Gremlin* graph traversal language [9], or Oracle's *PGX Algorithm* and *PGQL* need to be integrated into the client-server architecture. Typically, users send domain-specific code from the client to the server, where it is processed.

▶ **Example 1** (Graph analytics with *PGX*)**.** Figure 1 provides a simple interaction, in which a user compiles and runs a custom graph algorithm written in *PGX Algorithm*, before querying the result with a *PGQL* query. Figure 2 reproduces a similar interaction on a notebook. There, the query run by the user matches triangle patterns against graph data, and visualizes the result. Graph algorithm and query are processed on the server. Figure 3 illustrates the components of the *PGX* server which are involved in this process. *Spoofax*-implemented components are highlighted in gray.

The *PGX* server passes the algorithm code to the *PGX Algorithm* compiler for parsing, static analysis, optimization, and target code generation. The generated code is then compiled and loaded into the server. Finally, the server returns a symbolic handle of the compiled algorithm to the client. The user can then run the compiled algorithm. The server executes the algorithm and returns a summary of the execution to the client.

Similarly, the client sends *PGQL* queries to the server, where they are processed and executed. The *PGQL* compiler first parses the query, extracts the name of the queried graph, and looks up its metadata. Query and metadata are then statically analyzed for

Figure 1 Running a custom graph algorithm and a *PGQL* query in the *PGX Java* shell.

```
PgxGraph G = session.readGraphWithProperties("twitter.edge.json");
==> PgxGraph[name=Twitter,N=41652230,E=1468365182]

pgx> var compiled = session.compileProgram("DegreeCentrality.java");
==> CompiledProgram[name=degreeCentrality]

pgx> compiled.run(G, degree);
==> {"success" : true, "exception" : null, "returnValue" : null}

pgx> G.queryPgql("SELECT id(v), v.degree FROM MATCH (v) ORDER BY v.degree DESC LIMIT 3").
     print()
+------------------+
| id(v) | v.degree |
+------------------+
| 10009 | 22889    |
| 37356 | 15554    |
| 87    | 15218    |
+------------------+
```

name and type errors. Next, the compiler performs static optimizations on the query, before passing the query to the query optimizer. The query optimizer determines the most efficient execution plan of a query as a sequence of operators that can perform the necessary scanning, computation, and manipulation of graph data on the server. Finally, the result of the query is returned to the client.

Throughout the remainder of this section, we collect requirements for the integration of language processors into client-server architectures and report on our specific experiences with language processors derived from language definitions in *Spoofax*.

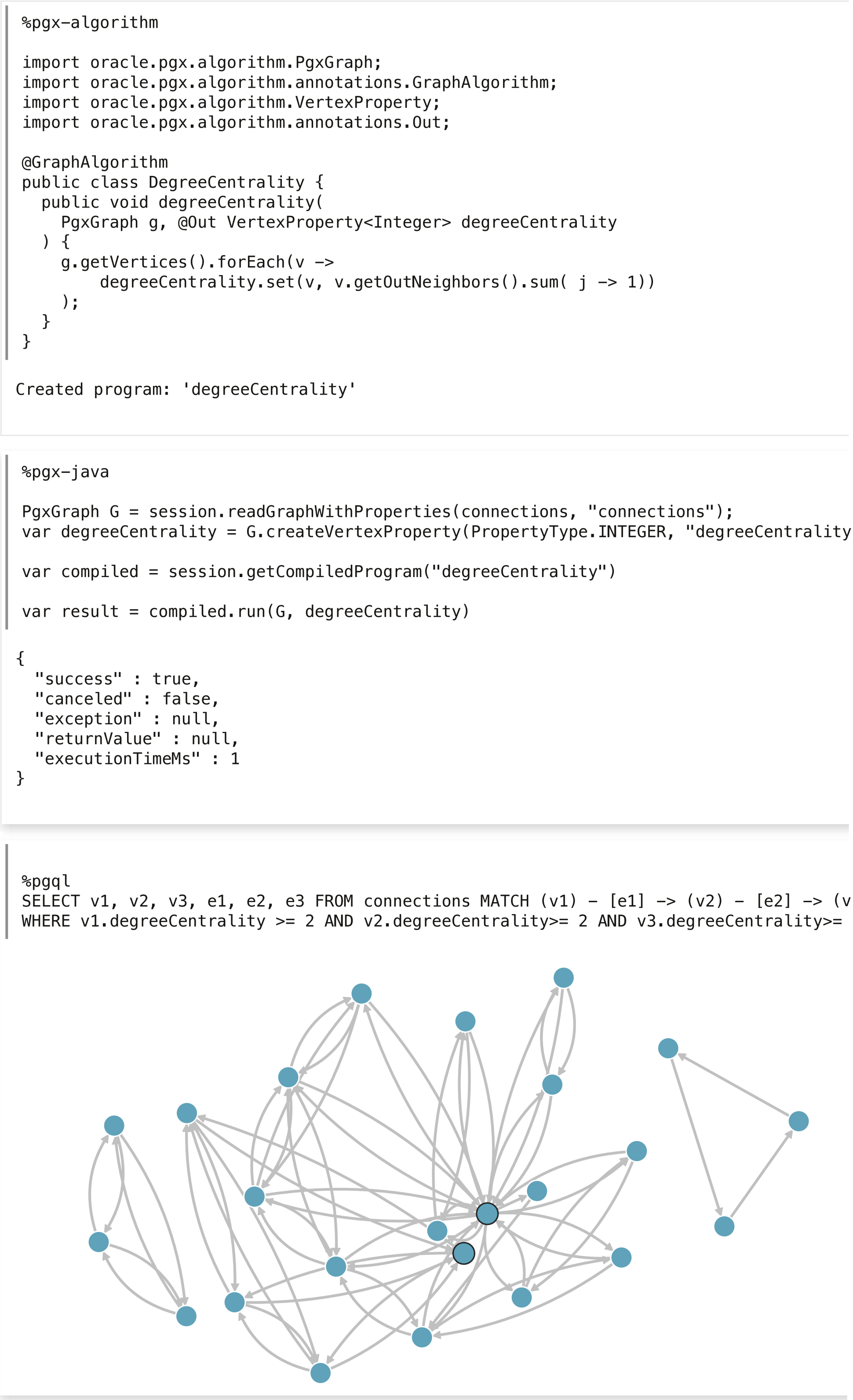
▶ **Requirement 1** (Programmatic Integration). *Language processors cannot be stand-alone tools but need to be programmatically integrated with the server.*

In its early stages, the main focus of the *Spoofax* project was to provide language processors for modern IDEs. Later, *Spoofax* also supported the generation of stand-alone language processors which could be invoked from command-line. Only *Spoofax 2* [3] introduced *Java* APIs to invoke language processors programmatically. We currently rely on this API to integrate the *PGX Algorithm* and *PGQL* compilers into the *PGX* server runtime.

▶ **Requirement 2** (Compliant Dependencies). *Language processors might depend on external software libraries. These dependencies need to comply with corporate policies for dependencies. Such policies might forbid particular versions of software libraries due to known vulnerabilities or entirely forbid the use of an external library in favor of internal solutions.*

PGX Algorithm and *PGQL* compilers depend on the *Spoofax* runtime, which itself has many dependencies on external software. Many of these dependencies are introduced to support IDE use cases, for example to load and reload language processors dynamically at runtime. This is problematic, since these dependencies are not required by our compilers, but still need to be regularly checked for compliance. The current work on *Spoofax 3* results in a notable decrease in dependencies.

▶ **Requirement 3** (Secure Language Processors). *Language processors run on the server and constitute potential targets for attacks. These can be Denial-of-Service attacks with extremely large or frequent requests to language processors. Language processors need to be embedded in a protective structure, which allows them to be interrupted when they hit a maximum processing time and to be shut down and restarted when they are in an exceptional state, for*



```
%pgx-algorithm

import oracle.pgx.algorithm.PgxGraph;
import oracle.pgx.algorithm.annotations.GraphAlgorithm;
import oracle.pgx.algorithm.VertexProperty;
import oracle.pgx.algorithm.annotations.Out;

@GraphAlgorithm
public class DegreeCentrality {
  public void degreeCentrality(
    PgxGraph g, @Out VertexProperty<Integer> degreeCentrality
  ) {
    g.getVertices().forEach(v ->
        degreeCentrality.set(v, v.getOutNeighbors().sum( j -> 1))
    );
  }
}
```

```
Created program: 'degreeCentrality'
```

```
%pgx-java

PgxGraph G = session.readGraphWithProperties(connections, "connections");
var degreeCentrality = G.createVertexProperty(PropertyType.INTEGER, "degreeCentrality

var compiled = session.getCompiledProgram("degreeCentrality")

var result = compiled.run(G, degreeCentrality)
```

```
{
  "success" : true,
  "canceled" : false,
  "exception" : null,
  "returnValue" : null,
  "executionTimeMs" : 1
}
```

```
%pgql
SELECT v1, v2, v3, e1, e2, e3 FROM connections MATCH (v1) - [e1] -> (v2) - [e2] -> (v1)
WHERE v1.degreeCentrality >= 2 AND v2.degreeCentrality>= 2 AND v3.degreeCentrality>= 2
```

Figure 2 Example of algorithm and query run on a notebook.

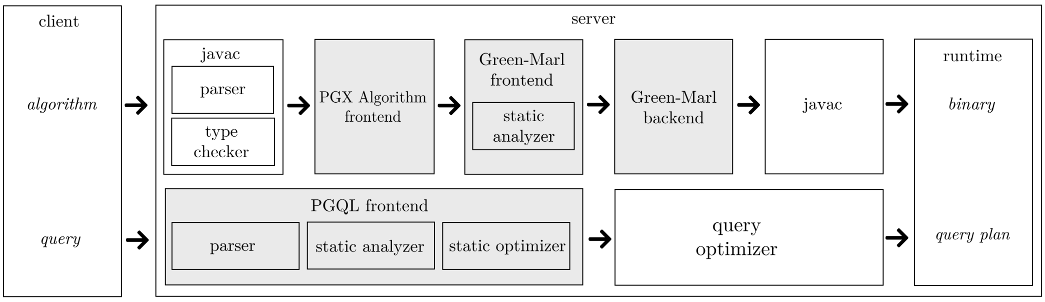


Figure 3 Language processing in the PGX server architecture.

example when running out of memory while processing a large input. The risk of attacks extends beyond language processors to the dependencies they rely on, such as dependencies for reading configurations or for logging[1]*.*

Spoofax' many dependencies increase the risk for vulnerabilities. The *Spoofax* team tracks known vulnerabilities, updates dependencies with fixed versions, and releases new versions of *Spoofax* on a regular basis, as well as on request from its industrial partners. A reduction in third-party dependencies is needed to reduce the risk of vulnerabilities in the *Spoofax* runtime.

▶ **Requirement 4** (Economic Memory Usage)**.** *Language processors typically run continuously to stay available for future requests throughout a server session. Thus, the memory footprint of idle language processors contribute to the overall memory footprint on the server.*

In our experience, *Spoofax* language processors tend to have larger memory footprints than needed, especially when language processors are generated from multiple dependent *Spoofax* projects. These dependencies tend to duplicate the code of the dependee into the dependent project, thus wasting memory. This causes language processors, specifically backends, to be unnecessarily big, as they include all the language definitions of the frontends they depend on, even if the latter are not needed by the backends at runtime. This will be solved in future releases of *Spoofax*, which rely on flexible, composable compiler pipelines.

3 Software (Language) Engineering for Graph Analytics

Domain-specific languages for graph analytics need to be designed, developed, maintained, tested, and released together with the larger graph analytics solution they belong to. Throughout this section, we collect requirements for the engineering of language processors as part of a larger software engineering project and report on our specific experiences with *Spoofax*.

▶ **Requirement 5** (Reusable Language Processors)**.** *Language processors need to be reusable in different settings. For example, a frontend might be combined with different backends for different products. Separated language processor projects allow for a combination of open-source and proprietary projects.*

Spoofax supports multi-project language definitions. We rely on this to split our language definitions into frontend and backend aspects. This allows us to compose language processors into different products. For example, we have separate projects for the *Green-Marl* frontend

[1] `https://nvd.nist.gov/vuln/detail/CVE-2021-44228`

and for each of its backends. We compose the frontend with one of the backends into a compiler we use internally. We recompose the frontend with another backend into another compiler for internal use. We compose the *PGX Algorithm* frontend, the *Green-Marl* frontend, and one of the *Green-Marl* backends into the *PGX Compiler* we include in Oracle products. Separate language projects also allow us to share some projects with our partners, while protecting proprietary solutions in internal projects. For example, the *PGQL* frontend is an open-source project, while the query optimizer is kept in an internal project. Similarly, we share the *Green-Marl* frontend project with academic collaboration partners. We also rely on existing open-source *Spoofax* projects. For example, the *PGX Algorithm* compiler integrates an open-source *Java* syntax definition in *Spoofax*. As discussed earlier, the memory footprint of composed language processors can be improved in future versions of *Spoofax*.

▶ **Requirement 6** (Accessible Language Implementations.). *Language implementations are often maintained by small teams. Team members might only contribute part-time or irregularly to the language development. New team members might join while others leave. This requires language implementations to be as accessible as possible. The source of a bug needs to be quickly found. An additional language construct needs to be easily added.*

Oracle adopted *Spoofax* for its promise of high-level, declarative, multi-purpose language definitions. After a decade of use, we find *Spoofax* delivering on this promise. The size of our language engineering teams fluctuates between a single part-time contributor and up to five full-time contributors. New team members get to know the code base quickly and can typically contribute small fixes after a couple of days and more complex features after a couple of weeks. To train new team members in the use of *Spoofax*, we rely on online material and technical support provided by the *Spoofax* team. Beside the declarative nature of language definitions in *Spoofax*, we find the possibility to modularize language definitions extremely useful to keep our codebase organized and thus easily accessible. We organize the code in hierarchical modules. This helps with maintenance, as the different language constructs are easily located in the project. It also enables the extension of the DSL and its compiler, since we can simply add modules for new language constructs, and have them import existing modules, without affecting the existing language implementation. This has allowed us to efficiently support new backends.

▶ **Requirement 7** (Early Development Feedback). *It is important to get early and quick feedback when implementing a new language feature or improving an existing implementation.*

Spoofax integrates into the *Eclipse* IDE and provides editors for its meta-languages with syntax highlighting, syntax checking, error marking, syntax completion, code formatting, type checking, reference resolving, and hover help. This helps to spot errors early, before building the language projects. Furthermore, the same services are available for the developed languages, allowing us to quickly test our languages in an IDE editor. *Spoofax* also provides menu actions for parsing and applying arbitrary transformations such as formatting, analyzing, and (partial) compilation. In our language projects, we configure several menu actions that perform (partial) compilations, which we trigger manually to quickly test the processing of small examples.

▶ **Requirement 8** (Language Processor Testing). *Adhoc tests can build some initial confidence in the implementation of a language processor, but systematic testing is needed to increase this confidence.*

Spoofax provides the *SPT* (SPoofax Testing) meta-language to specify tests for language definitions. We rely on *SPT* to write unit tests for parsing, static analysis, and compilation. These unit tests typically cover each language construct in separate tests. We organize tests in separate projects, following the project structure and the hierarchical module structure of our compilers. We also use *SPT* to write integration tests in order to ensure complex queries and algorithms are correctly processed. However, these tests only address the language processors as stand-alone tools, and do not consider their integration with larger graph analytics solutions.

To test this integration, we have several tests which compile and run queries and algorithms as part of a toolkit or product. Product integration tests call the compiler for a given runtime and execute a number of algorithms in that runtime, checking for unexpected behavior in the compiler, errors while compiling the generated code into binaries, errors while executing the binary in the runtime, and unexpected results of the executed algorithm. These tests check behavior of built-in algorithms, compilation of custom algorithms, and (in)compatibility of certain language features with specific runtimes.

Customers rely on the performance of built-in and custom algorithms. As such, every change to language implementations or runtimes must be tested for degrading performance. Integration and performance tests are generally hardware-intensive to run, and may require a certain client-server architecture. We run them in a cluster, as part of a regression build triggered when a change is submitted to one of our code repositories. Only when this build succeeds can the change be committed.

▶ **Requirement 9** (Automated Builds). *Continuous development and integration of language processors require automated builds of language processors. These builds need to take dependencies between language projects into account. Tests need to be run as part of regression builds. All automatic builds need to work for stand-alone language processors, but also need to be integrated into overarching builds of toolkits and products.*

Early versions of *Spoofax* mainly focused on the interactive language development experience in IDEs. The building steps for *Spoofax* languages were soon encapsulated in *Maven* builds, which then also could be used to build language projects from command-line tools and to automate builds. However, automated *Spoofax* language builds often felt fragile, and hard to get right for new project setups, particularly for multi-project setups. We currently rely on an intermediate solution to integrate *Spoofax* language builds into our *Gradle* builds. The current work on flexible, incremental compilation pipelines is an important milestone towards improving the automated build situation.

▶ **Requirement 10** (Product Releases). *Language implementations need to be maintained as part of long-term support releases of a product. This includes bug fixes, feature backports, and dependency updates.*

The *Spoofax* team provides multiple stable releases of *Spoofax* per year, but none of these releases provide long-term support. This impacts the long-term support of Oracle products. If language processors in a product are affected by bugs or vulnerabilities introduced by *Spoofax* or its dependencies, we need to update these language processors to work with the latest stable *Spoofax* release. Such an update can be problematic if the *Spoofax* release introduces breaking changes.

4 Conclusion

In this paper, we looked back on how teams at Oracle used *Spoofax* to develop domain-specific languages for large-scale graph analytics for over a decade. We discussed the requirements for language processors in the domain of graph analytics and for their development processes. Our experience and the resulting requirements have motivated some of the development directions of *Spoofax*. *Spoofax* continues to help us to explore language designs, optimization techniques, and compilation patterns in various research projects at Oracle Labs, but also supports us in evolving early research prototypes into solid language processors which become part of Oracle products.

References

1 Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2150976.2151013`.

2 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 237–238, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1869542.1869592`.

3 Gabriël Konat. *Language-Parametric Methods for Developing Interactive Programming Systems*. PhD thesis, Delft University of Technology, Delft, Netherlands, November 2019.

4 Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745, pages 311–331, January 2013. `doi:10.1007/978-3-642-36089-3_18`.

5 Neo4j. Cypher query language, 2022. URL: `https://neo4j.com/developer/cypher/`.

6 Oracle. Pgql property graph query language, 2022. URL: `https://pgql-lang.org`.

7 Oracle. Pgx documentation, 2022. URL: `https://docs.oracle.com/cd/E56133_01/latest/reference/analytics/pgx-algorithm.html`.

8 Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. Using domain-specific languages for analytic graph databases. *Proc. VLDB Endow.*, 9(13):1257–1268, September 2016. `doi:10.14778/3007263.3007265`.

9 Tinkerpop. Tinkerpop, gremlin, 2022. URL: `https://github.com/tinkerpop/gremlin/wiki`.

10 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2960414.2960421`.

11 Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, August 1997.

12 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

13 Guido H. Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 260–280, Cham, 2013. Springer International Publishing.

Dependently Typed Languages in Statix

Jonathan Brouwer
Delft University of Technology, The Netherlands

Jesper Cockx
Delft University of Technology, The Netherlands

Aron Zwaan
Delft University of Technology, The Netherlands

Abstract

Static type systems can greatly enhance the quality of programs, but implementing a type checker that is both expressive and user-friendly is challenging and error-prone. The Statix meta-language (part of the Spoofax language workbench) aims to make this task easier by automatically deriving a type checker from a declarative specification of a type system. However, so far Statix has not been used to implement dependent types, which is a class of type systems which require evaluation of terms during type checking. In this paper, we present an implementation of a simple dependently typed language in Statix, and discuss how to extend it with several common features such as inductive data types, universes, and inference of implicit arguments. While we encountered some challenges in the implementation, our conclusion is that Statix is already usable as a tool for implementing dependent types.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Functional languages; Software and its engineering → Compilers

Keywords and phrases Spoofax, Statix, Dependent Types, Scope Graphs, Calculus of Constructions

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.6

Supplementary Material *Software (Source Code)*: https://doi.org/10.5281/zenodo.7541014

1 Introduction

Spoofax is a language workbench: a collection of tools that enable the development of textual languages [7]. When working with the Spoofax workbench, the Statix meta-language can be used for the specification of static semantics. It is a declarative language that uses inference rules to define static semantics. From these rules a type checker and other editor tools can be derived. Statix aims to cover a broad range of languages and type systems. However, no attempts have been made to express dependently typed languages in Statix until now.

Dependently typed languages are different from other languages, because they allow types to be parameterized by values. This allows types to express properties of values that cannot be expressed in a simple type system, such as the length of a list or the well-formedness of a binary search tree. This expressiveness also makes dependent type systems more complicated to implement. Especially, deciding equality of types requires normalization of the terms they are parameterized by.

This goal of this paper is to investigate how well Statix is fit for the task of defining a simple dependently-typed language. We want to investigate whether typical features of dependently typed languages can be encoded concisely in Statix. The goal is not only to show that Statix can implement it, but also that the implementation is concise.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 6; pp. 6:1–6:8
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In section 2 of this paper we show that Statix is already usable as a tool for implementing dependent types. There were some challenges in implementing the dependent type system, which this paper also discusses. Then, in section 3 we describe how to extend the language with several common features such as inductive data types, universes, and inference of implicit arguments.

2 Calculus of Constructions

In this section, we will describe how to implement a dependently typed language in Statix. In section 2.1 we will describe the syntax of the language, then in section 2.2 we will describe how scope graphs are used to type check the language. Section 2.3 describes the dynamic semantics of the language, and finally 2.4 how to type check the language.

2.1 The language

The base language that has been implemented is the Calculus of Constructions [4]. One extra feature was added that is not present in the Calculus of Constructions: let bindings. Let bindings could be desugared by substituting, but this may grow the program size exponentially, so having them in the language is useful. The concrete syntax (written in SDF3 [5]) of the language is available in figure 1.

```
Expr.Let = "let" ID "=" Expr ";" Expr
Expr.Type = "Type"
Expr.Var = ID
Expr.FnType = ID ":" Expr "->" Expr {right}
Expr.FnConstruct = "\\" ID ":" Expr "." Expr
Expr.FnDestruct = Expr Expr {left}
```

Figure 1 The concrete syntax for the base language. FnConstruct is a lambda function, FnDestruct is application of a lambda function.

Please observe that the syntax definition does not have a separate sort for types, as types may be arbitrary expressions in a dependently typed language. Furthermore, `FnType` assigns a name to its first argument to allow the return type of the function to depend on the argument type.

An example program is the following, which defines a polymorphic identity function and applies it to a function:

```
let f = \T: Type. \x: T. x;
f (T: Type -> Type) (\y: Type. y)
```

2.2 Scope Graphs

To type check the base language, we need to store information about the names that are in scope at each point in the program. There are two different cases, names that do not have a known value (only a type), which are names created by function arguments and dependent function types, and names that do have a known value, which are names created by let bindings.[1]

[1] In non-dependent languages there is no such distinction, but because we may need *the value* of a binding to compare types, this is needed in dependently typed languages.

In Statix, all this information can be stored in a *scope graph* [11]. It is a graph consisting of nodes for scopes, labeled edges for visibility relations, scoped declarations for a relation, and queries for references. We only use a single type of edge, called `P` (parent) edges. It also only has a single relation, called `name`. This name stores a `NameEntry`, which can be either a `NType`, which stores the type of a name, or a `NSubst`, which stores a name that has been substituted with a value. The Statix definition of these concepts is given below:

```
constructors
    NameType : Expr -> NameEntry
    NameSubst : scope * Expr -> NameEntry
relations
    name : ID -> NameEntry
```

Next, we will introduce some Statix predicates that can be used to interact with these scope graphs:

```
sPutType  : scope * ID * Expr -> scope
sPutSubst : scope * ID * (scope * Expr) -> scope
sGetName  : scope * ID -> NameEntry
sEmpty    : -> scope
```

The `sPutType` and `sPutSubst` predicates generate a new scope given a parent scope and a type or a substitution respectively. To query the scope graph use `sGetName`, which will return a `NameEntry` that the query found. Finally, `sEmpty` returns a fresh empty scope.

We will define a *scoped expression*, as a pair of a scope and an expression. The scope acts as the environment of the expression, containing all of the context needed to evaluate the expression.

2.3 Beta Reductions

A unique requirement for dependently typed languages is beta reduction during type checking, since types may require evaluation to compare. Beta reduction is the process of reduction a term to its beta normal form, which is the state where no further beta reductions are possible [15]. It works by matching a term of the form `(\x. b) e.` and substituting `x` in `b` with `e`. Beta reduction applies this rule anywhere in the term, whereas beta head-reduction only applies this rule at the head (outermost expression) of the term, and produces a term in beta-head normal form.

We implemented beta reduction using a Krivine abstract machine [8]. The machine can head evaluate lambda expressions with a call-by-name semantics. It works by keeping a stack of all arguments that have not been applied yet. This turned out to be the more natural way of expressing this compared to substitution-based evaluation relation. We originally tried to implement the latter, which works fine for the base language. However, it runs into trouble when implementing inductive data types; more information about this will be in the full master's thesis [3]. An additional benefit is that abstract machines are usually more efficient than substitution-based approaches.

In conventional dependently typed languages, evaluation is often done using De Bruijn indices. However, we chose to use names rather than De Bruijn indices, because scope graphs work based on names. Using De Bruijn indices would also prevent us from using editor services that rely on `.ref` annotations (which are Spoofax annotations that declare one name as being a use of another name that is a definition), such as renaming.

We need to define multiple predicates that will be used later for type checking. First, the primary predicate is `betaReduceHead`, that takes a scoped expression and a stack of applications, and returns a head-normal expression. The scope acts as the environment from [8], using

NSubst to store substitutions. All rules for betaReduceHead are given in figure 2. We use the syntax $\langle s_1 \mid e_1\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2\rangle$ to express `betaReduceHead((s1, e1), ps) == (s2, e2)`. The `p` in this definition is the argument stack of the Krivine machine. Figure 2 contains the rules necessary for beta head reduction of the language. One predicate that is used for this is the `rebuild` predicate, which takes a scoped expression and the stack of arguments (of the Krivine machine state) and converts it to an expression by adding `FnDestructs`. Its signature is:

```
rebuild : (scope * Expr) * list((scope * Expr)) -> (scope * Expr)
```

Additionally, we define `betaReduce` which fully beta reduces a term. It works by first calling `betaReduceHead` and then matching on the head, calling `betaReduce` on the sub-expressions of the head recursively.

Finally, we define `expectBetaEq`. This rule first beta reduces the heads of both sides, and then compares them. If the head is not the same, the rule fails. Otherwise, it recurses on the sub-expressions. One special case is when comparing two `FnConstructs`. Here we need to take into account alpha equality: two expressions which only differ in the names that they use should be considered equal. We implement this by substituting in the body of the functions, replacing their argument names with placeholders.

Beta head-reduction rules $\boxed{\langle s_1 \mid e_1\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2\rangle}$

$$\frac{}{\langle s \mid \mathsf{Type}()\rangle\, [\,] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{Type}()\rangle} \qquad \frac{\langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle}{\langle s \mid \mathsf{Let}(x, e, b)\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e'\rangle}{\langle s \mid \mathsf{Var}(x)\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e'\rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NType}(t)}{\langle s \mid \mathsf{Var}(x)\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \mathsf{rebuild}(s, \mathsf{Var}(x), \overline{p})} \qquad \frac{}{\langle s \mid \mathsf{FnType}(x, a, b)\rangle\, [\,] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{FnType}(x, a, b)\rangle}$$

$$\frac{}{\langle s \mid \mathsf{FnConstruct}(x, a, b)\rangle\, [\,] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{FnConstruct}(x, a, b)\rangle}$$

$$\frac{\langle \mathsf{sPutSubst}(s, x, p) \mid b\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle}{\langle s \mid \mathsf{FnConstruct}(x, \_, b)\rangle\, (p :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle} \qquad \frac{\langle s \mid f\rangle\, (a :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle}{\langle s \mid \mathsf{FnDestruct}(f, a)\rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle}$$

Figure 2 Rules for beta head reducing the Calculus of Constructions.

2.4 Type checking the Calculus of Constructions

We will define a Statix predicate `typeOfExpr` that takes a scope and an expression and type checks the expression in the scope. It returns the type of the expression.

```
typeOfExpr : scope * Expr -> Expr
```

We can then start defining type checking rules for the language. We introduce a number of judgements for typing and equality together with their counterparts in Statix.

1. $\langle s \mid e \rangle : t$ is the same as `typeOfExpr(s, e) == t`
2. $\langle s1 \mid e1 \rangle \underset{\beta}{=} \langle s2 \mid e2 \rangle$ is the same as `expectBetaEq((s1, e1), (s2, e2))`
3. $\langle s1 \mid e1 \rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s2 \mid e2 \rangle$ is the same as `betaReduceHead((s1, e1), ps) == (s2, e2)` (The same as in section 2.3)
4. $\langle s1 \mid e1 \rangle \underset{\beta}{\Rightarrow} e2$ is the same as `betaReduce((s1, e1)) == e2`
5. $\langle \mathit{sEmpty} \mid e \rangle$ is the same as e (empty scopes can be left out)

The inference rules in figure 3 can be directly translated to Statix rules. For example, the rule for `Let` bindings is expressed like this in Statix:

```
typeOfExpr(s, Let(n, v, b)) = typeOfExpr(s', b) :-
    typeOfExpr(s, v) == vt, sPutSubst(s, n, (s, v)) == s'.
```

Type checking rules $\boxed{\langle s \mid e \rangle : t}$

$$\frac{}{\langle s \mid \mathsf{Type}() \rangle : \mathsf{Type}()} \qquad \frac{\langle s \mid e \rangle : t_e \qquad \langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b \rangle : t_b}{\langle s \mid \mathsf{Let}(x, e, b) \rangle : t_b}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NType}(t)}{\langle s \mid \mathsf{Var}(x) \rangle : t} \qquad \frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e \rangle : t}{\langle s \mid \mathsf{Var}(x) \rangle : t}$$

$$\frac{\begin{array}{ccc} \langle s \mid a \rangle : t_a & t_a \underset{\beta}{=} \mathsf{Type}() & \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \\ \multicolumn{2}{c}{\langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b} & t_b \underset{\beta}{=} \mathsf{Type}() \end{array}}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle : \mathsf{Type}()} \qquad \frac{\begin{array}{ccc} \langle s \mid a \rangle : t_a & t_a \underset{\beta}{=} \mathsf{Type}() & \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \\ \multicolumn{3}{c}{\langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b} \end{array}}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle : \mathsf{FnType}(x, a', t_b)}$$

$$\frac{\begin{array}{c} \langle s \mid f \rangle : t_f \qquad \langle s \mid t_f \rangle\, [] \underset{\beta h}{\Rightarrow} \langle s_f \mid \mathsf{FnType}(x, t_{da}, t_b) \rangle \\ \langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \langle s_f \mid t_{da} \rangle \qquad \langle \mathsf{sPutSubst}(s_f, x, (s, a)) \mid t_b \rangle \underset{\beta}{\Rightarrow} t_b' \end{array}}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle : t_b'}$$

Figure 3 Rules for type checking the Calculus of Constructions.

2.5 Avoiding variable capture

One problem with the implementation presented in the previous sections is that it does not avoid variable capture. Variable capture happens when a term becomes bound because of a wrong substitution. For example, according to the inference rules in figure 3 the type of `\T : Type. \T : T. T` is `T : Type -> T : T -> T`. This is not the correct type, and even worse, it is not possible to express the correct type without renaming! The problem is that there are multiple names, and there is no way to distinguish between them. This problem needs to be addressed in any name-based approach. It can be solved by using scopes to distinguish names. A full explanation of this will be in the master's thesis [3].

3 Extensions

This section will discuss some further ideas that can be explored to build upon what has already been shown. These will be fully described in the master's thesis [3] that this paper is based on, this is just an exploration of what is possible.

Term Inference

In dependently typed languages, the value of types can often be inferred. Ideally, we would like to infer the most general type possible. However, this kind of analysis is not possible in Statix because you cannot reason over whether meta-variables are instantiated or not. We implemented an approximation to inference that can infer most types. For example, the type of `x` in the function below can be inferred, because it is applied to `true`, which is a boolean.

```
(\x: _. x) true
```

Inductive Data Types

Another common feature in dependently typed language is support for inductive data types, including support for parameterized and indexed data types. We can also generate eliminators for these data types to use their values. All of this has been implemented in Statix.

```
data Maybe (T : Type) =
    None : Maybe T,
    Some : x: T -> Maybe T;
```

Semantic Code Completion

Finally, we explored how semantic code completion presented by Pelsmaeker et al. [12] applies to dependently typed languages. It works well, only showing completions that are semantically possible.

For example, in the following code it would only suggest expressions that can be booleans:

```
let f = \b: Bool. b;
f [[Expr]]
```

This would return the following suggestions: (Note that `f` and `Type` are not suggested, since they cannot have type boolean)

- `[[Expr]] [[Expr]]`
- `let [[ID]] = [[Expr]]; [[Expr]]`
- `true`
- `false`
- `if [[Expr]] then [[Expr]] else [[Expr]] end`

4 Related Work

The implementation in this paper requires performing substitutions in types immediately, as types don't have a scope. Van Antwerpen et al. [16, sect 2.5] present an implementation of System F that does lazy substitutions, by using scopes as types. It would be interesting to see if this approach could also apply to the Calculus of Constructions, where types can contain terms.

Another interesting comparison is to see how implementing a dependently typed language in Statix differs from implementing it in a general purpose language. The pi-forall language [17] is a good example of a language with a similar complexity to the language presented in this paper. In principle, the implementations are very similar. For example, the inference rules of pi-forall are similar to the inference rules presented in figure 3 from this paper. The primary difference is that they use a bidirectional type system [6], whereas this paper uses Statix' unification.

There exist several so-called *logical frameworks*, tools designed specifically for implementing and experimenting with dependent type theories, such as ALF [9], Twelf [14], Dedukti [2], Elf [13] and Andromeda [1]. Since these tools are designed specifically for the task, implementing the type system takes less effort in them compared to Spoofax, but for other tasks such as defining a parser or editor services they are not as well equipped. Some logical frameworks such as Twelf and Dedukti support Miller's *higher-order pattern unification* [10], which can be used as a more powerful way of inferring implicit arguments than the first-order unification built into Statix. Andromeda 2 also supports *extensionality rules* that can match on the type of an equality. We expect that adding extensionality rules to our implementation would be possible to do in Statix, but we leave an actual implementation to future work.

5 Conclusion

We have demonstrated that the Calculus of Constructions can be implemented concisely in Statix, by storing substitutions in the scope graph. We have also presented a few extensions to the Calculus of Constructions and discussed how they could be implemented.

References

1 Andrej Bauer, Philipp G. Haselwarter, and Anja Petkovic. Equality checking for general type theories in andromeda 2. In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff, editors, *Mathematical Software – ICMS 2020 – 7th International Conference, Braunschweig, Germany, July 13-16, 2020, Proceedings*, volume 12097 of *Lecture Notes in Computer Science*, pages 253–259. Springer, 2020. doi:10.1007/978-3-030-52200-1_25.

2 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The lm-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving, PxTP 2012, Manchester, UK, June 30, 2012*, volume 878 of *CEUR Workshop Proceedings*, pages 28–43. CEUR-WS.org, 2012. URL: http://ceur-ws.org/Vol-878/paper2.pdf.

3 Jonathan Brouwer, Jesper Cockx, and Aron Zwaan. Dependently typed languages in statix. Delft University Of Technology. To be published on https://repository.tudelft.nl.

4 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, February 1988. doi:10.1016/0890-5401(88)90005-3.

5 Luís Eduardo de Souza Amorim and Eelco Visser. Multi-purpose syntax definition with sdf3. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings*, pages 1–23, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-58768-0_1.

6 Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021. doi:10.1145/3450952.

7 Lennart Kats and Eelco Visser. The spoofax language workbench. *ACM SIGPLAN Notices*, 45:237–238, October 2010. doi:10.1145/1869542.1869592.

8 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007. doi:10.1007/s10990-007-9018-9.

9 Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES 93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer, 1993. doi:10.1007/3-540-58085-9_78.

10 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming, International Workshop, Tübingen, FRG, December 8-10, 1989, Proceedings*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer, 1989. doi:10.1007/BFb0038698.

11 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems – 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

12 Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), April 2022. doi:10.1145/3527329.

13 Frank Pfenning. *Logic programming in the LF logical framework*, pages 149–182. Cambridge University Press, 1991. doi:10.1017/CBO9780511569807.008.

14 Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999. URL: http://link.springer.de/link/service/series/0558/bibs/1632/16320202.htm.

15 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, February 2002.

16 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018. doi:10.1145/3276484.

17 Stephanie Weirich. Implementing dependent types in pi-forall, 2022. doi:10.48550/arXiv.2207.02129.

Generating Software for Well-Understood Domains

Jacques Carette
Department of Computing and Software, McMaster University, Hamilton, Canada

Spencer W. Smith
Department of Computing and Software, McMaster University, Hamilton, Canada

Jason Balaci
Department of Computing and Software, McMaster University, Hamilton, Canada

Abstract

Current software development is often quite code-centric and aimed at short-term deliverables, due to various contextual forces (such as the need for new revenue streams from many individual buyers). We're interested in software where different forces drive the development. **Well understood domains** and **long-lived software** provide one such context.

A crucial observation is that software artifacts that are currently handwritten contain considerable duplication. By using domain-specific languages and generative techniques, we can capture the contents of many of the artifacts of such software. Assuming an appropriate codification of domain knowledge, we find that the resulting de-duplicated sources are shorter and closer to the domain. Our prototype, Drasil, indicates improvements to traceability and change management. We're also hopeful that this could lead to long-term productivity improvements for software where these forces are at play.

2012 ACM Subject Classification Software and its engineering → Application specific development environments; Software and its engineering → Requirements analysis; Software and its engineering → Specification languages; Software and its engineering → Automatic programming

Keywords and phrases code generation, document generation, knowledge capture, software engineering

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.7

Supplementary Material *Software (Source Code)*: `https://github.com/JacquesCarette/Drasil` archived at `swh:1:dir:5172b61ad2097f1fcbc50bf4828876f44fc22eb0`

1 The Context

Not all software is the same. In fact, there is enough variation in the context in which developers create various software products to warrant exploring and using different processes, depending on the forces [1] at play. Here we explore two such forces: "well understood" and "long lived".

We need to be precise about what we mean by "well understood" and "long lived", and thus will start by defining these terms. The definition of "well understood" has three components, which will be successively refined throughout this paper. As the codification of what is understood tends to be reliably communicated via *documentation*, we review that as well, before discussing which aspects of software products, aka software artifacts, we are considering. Then we give concrete examples of software domains where these ideas come together.

This work can be seen as resurrecting many of the goals of projects like Draco [21, 22], DMS [5, 6], GLisp [23, 24] and SpecWare [32, 30]. All are variants on *automatic programming*. Unlike many of these projects, our primary aim is not automation, but rather a combination of raising the level of abstraction and knowledge capture. And indeed *domain engineering*, as coined by Draco's creator James Neighbors, is core to our work. Where we differ is our

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 7; pp. 7:1–7:12
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

emphasis on all software artifacts, not just code, and being able to leverage several decades of theoretical and technological advances. See 4 for more details on these and other related projects and ideas.

1.1 "Well-understood" software?

▶ **Definition 1.** *A software domain is* ***well understood*** *if*
1. *its Domain Knowledge (DK) [8] is codified,*
2. *the computational interpretation of the DK is clear, and*
3. *writing code to perform said computations is well understood.*

By *codified*, we mean that the knowledge exists in standard form in a variety of textbooks. For example, many engineering domains use ordinary differential equations as models, the quantities of interest are known, given standard names and standard units. In other words, standard vocabulary has been established over time and the body of knowledge is uncontroversial.

We can refine these high level ideas, using the same numbering, although the refinement should be understood more holistically.
1. Models in the DK *can be* written formally.
2. Models in the DK *can be* turned into functional relations by existing mathematical steps.
3. Turning these functional relations into code is an understood transformation.

Most importantly, the last two parts deeply involve *choices*: What quantities are considered inputs, outputs and parameters to make the model functional? What programming language? What software architecture, data-structures, algorithms, etc.?

In other words, *well understood* does not imply *choice free.* Writing a small script to move files could just as easily be done in Bash, Python or Haskell. In all cases, assuming fluency, the author's job is straightforward because the domain is well understood.

1.2 Long-lived software?

For us, long-lived software [18] is software that is expected to be in continuous use and evolution for 20 or more years. The main characteristic of such software is the *expected turnover* of key staff. This means that all tacit knowledge about the software will be lost over time if it is not captured. Relying on oral tradition and word-of-mouth becomes untenable for the viability of long-lived projects, and documentation becomes a core asset of value comparable, if not greater than, the actual code.

1.3 Documentation

Well understood also applies to **documentation** aimed at humans [26]. The same three items of Definition 1 can be retargeted to apply to documentation as follows: (1) The meaning of the models is understood at a human-pedagogical level, i.e., it is explainable. (2) Combining models is explainable. Thus, the act of combining models must simultaneously operate on mathematical representations and on explanations. This requires that English descriptions also be captured in the same manner as the formal-mathematical knowledge. (3) The refinement steps that are performed due to making software-oriented decisions should be captured with a similar mechanism, and also include English explanations.

Coupling together well-understood domains and its documentation, we obtain what we call triform theories.

▶ **Definition 2.** ***Triform theories****, as a nod to* biform theories *[11], are the coupling of three concepts:*

1. *an axiomatic description,*
2. *a computational description, and*
3. *an English description.*

These will form the heart of our approach to domain engineering.

1.4 Software artifacts

Software currently consists of a whole host of artifacts: requirements, specifications, user manual, unit tests, system tests, usability tests, build scripts, READMEs, license documents, process documents, as well as code. Our aim is to understand the "information content" of each artifact sufficiently to construct a Domain Specific Language (DSL) that describes how to weave information gleaned from triform theories with instance specific choices that are sufficient to enable the generation of each artifact.

1.5 Instances

When are these well understood and long lived conditions fulfilled? One example is *research software* in science and engineering. While the results of running various simulations is entirely new, the underlying models and how to simulate them are indeed well known. One particularly long-lived example is embedded software for space probes (like Pioneer 10). Another would be ocean models, such as NEMO [12], which can trace its origins to OPA 8 [20]. In fact, most subdomains of science and engineering harbour their own niche long-lived software. The domain of control systems is particularly rich in examples.

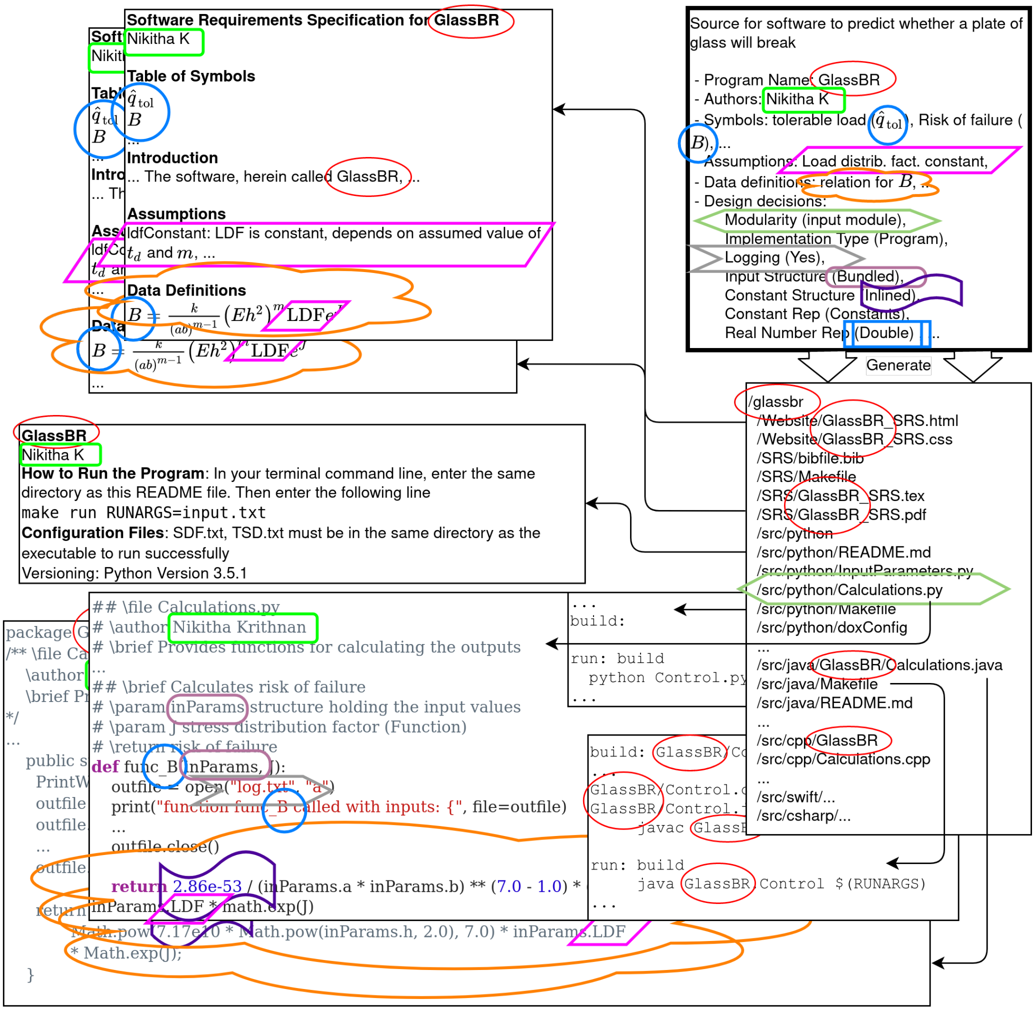
Note that some better-known long-lived software, such as financial systems, do not fall within our purview, as the implemented business processes are not *well understood*. This is why most large rewrites of financial systems fail as the functionality actually implemented in the source system is not documented.

2 An Example

We have built infrastructure, which we also call Drasil[1] to carry out these ideas. It consists of 60KLoc of Haskell implementing a series of interacting Domain Specific Languages (DSLs) for knowledge encodings, mathematical expressions, theories, English fragments, code generation and document generation.

Drasil consists of a "database" of information about certain domains of interest (parts of physics, some mathematics, software engineering, documentation and computing), encoded in a home-grown ontology somewhat resembling the work of Novak [16, 24]. It has also a number of internal DSLs for representing the contents of various kinds of documents (specifications, programs, simple build scripts, READMEs), coupled with DSLs for rendering in various formats (LaTeX, HTML and plain text, used for specifications, and GOOL [19] for representing Object-Oriented programs that can be rendered in six different languages). There are also DSLs for triform theories, mathematical expressions, English fragments, citations, authors, some software choices, and so on. Lastly another DSL weaves together all the previous information to render the artifacts that together make a piece of software. A longer article describing all of these in detail is being written.

[1] https://github.com/JacquesCarette/Drasil



Software Requirements Specification for GlassBR
Nikitha K

Table of Symbols
$\hat{q}_{\text{tol}}$
B
...

Introduction
... The software, herein called GlassBR, ...

Assumptions
ldfConstant: LDF is constant, depends on assumed value of t_d and m, ...

Data Definitions
$B = \frac{k}{(ab)^{m-1}} (Eh^2)^m \text{LDF} e^J$
...

Source for software to predict whether a plate of glass will break

- Program Name: GlassBR
- Authors: Nikitha K
- Symbols: tolerable load ($\hat{q}_{\text{tol}}$), Risk of failure (B), ...
- Assumptions: Load distrib. fact. constant,
- Data definitions: relation for B, ...
- Design decisions:
  Modularity (input module),
  Implementation Type (Program),
  Logging (Yes),
  Input Structure (Bundled),
  Constant Structure (Inlined),
  Constant Rep (Constants),
  Real Number Rep (Double) ...

```
/glassbr
  /Website/GlassBR_SRS.html
  /Website/GlassBR_SRS.css
  /SRS/bibfile.bib
  /SRS/Makefile
  /SRS/GlassBR_SRS.tex
  /SRS/GlassBR_SRS.pdf
  /src/python
  /src/python/README.md
  /src/python/InputParameters.py
  /src/python/Calculations.py
  /src/python/Makefile
  /src/python/doxConfig
  ...
  /src/java/GlassBR/Calculations.java
  /src/java/Makefile
  /src/java/README.md
  ...
  /src/cpp/GlassBR
  /src/cpp/Calculations.cpp
  ...
  /src/swift/...
  /src/csharp/...
```

GlassBR
Nikitha K
How to Run the Program: In your terminal command line, enter the same directory as this README file. Then enter the following line
`make run RUNARGS=input.txt`
Configuration Files: SDF.txt, TSD.txt must be in the same directory as the executable to run successfully
Versioning: Python Version 3.5.1

```
...
build:

run: build
  python Control.py
...
```

```
build: GlassBR/Co
...
GlassBR/Control.c
GlassBR/Control.
    javac GlassB

run: build
    java GlassBR.Control $(RUNARGS)
...
```

```
## \file Calculations.py
# \author Nikitha Krithnan
# \brief Provides functions for calculating the outputs
...
## \brief Calculates risk of failure
# \param inParams structure holding the input values
# \param J stress distribution factor (Function)
# \return risk of failure
def func_B(inParams, J):
    outfile = open("log.txt", "a")
    print("function func_B called with inputs: {", file=outfile)
    ...
    outfile.close()

    return 2.86e-53 / (inParams.a * inParams.b) ** (7.0 - 1.0) *
inParams.LDF * math.exp(J)
```

```
package G
/** \file Ca
  \author
  \brief Pr
*/
...
  public s
    PrintW
    outfile
    outfile
    ...
    outfile
    return
      Math.pow(7.17e10 * Math.pow(inParams.h, 2.0), 7.0) * inParams.LDF
      * Math.exp(J);
  }
```

Figure 1 Colors and shapes mapping from captured domain knowledge to generated artifacts. Red oval: program name; green rectangle: author name; blue circle: symbols for real-valued variables; pink lozenge: assumptions used (load distribution factor is constant); orange cloud: mathematical definition of B; gray arrow: option to turn on logging; light brown rounded-corners rectangle: how inputs should be “bundled”; purple wavy rectangle: how constants are handled (here: inlined).

We provide a bit of the overall flavour of Drasil through what we hope to be an illustrative example.

2.1 GlassBR

We will focus on information capture and the artifacts we can generate. For concreteness, we’ll use a single example from our suite: GlassBR, used to assess the risk for glass facades subject to blast loading. The requirements are based on an American Standard Test Method (ASTM) standard [3, 4, 7]. GlassBR was originally a Visual Basic code/spreadsheet created by colleagues in a civil engineering research lab. We added their domain knowledge to our framework, along with recipes to generate relevant artifacts. Not only can we generate code for the necessary calculations (in C++, C#, Java, Python and Swift), we also generated documentation that was absent in the original (Software Requirements Specification, doxygen, README.md and a Makefile). Moreover, our implementation is actually a family of implementations, since some design decisions are explicitly exposed as changeable variabilities, as described below.

Figure 1 illustrates the transformation of captured domain knowledge. Reading this figure starts from the upper right box. Each piece of information has its own shape and colour (orange cloud, pink lozenge, etc). It should be immediately clear that all pieces of information reappear in multiple places in the generated artifacts. For example, the name of the software (GlassBR) ends up appearing more than 80 times in the generated artifacts (in the folder structure, requirements, README, Makefile and source code). Changing this name would traditionally be extremely difficult; we can achieve this by modifying a single place, and regenerating.

The first box shows the directory structure of the currently generated artifacts; continuing clockwise, we see examples of Makefiles for the Java and Python versions, parts of the fully documented, generated code for the main computation in those languages, user instructions for running the code, and the processed LaTeX for the requirements.

The name GlassBR is probably the simplest example of what we mean by *duplication*: here, the concept "program name" is internally defined, and its *value* is used throughout.

In general, we capture more complex knowledge. An example is the assumption that the "Load Distribution Factor" (LDF) is constant (pink lozenge). If this needs to be modified to instead be an input, the generated software will now have LDF as an input variable.

We also capture design decisions, such as whether to log all calculations, whether to inline constants rather than show them symbolically, etc. These different pieces of knowledge can also be reused in different projects.

2.2 The Steps

We describe an "idealized process" that we could have used to produce GlassBR, following Parnas' idea of faking a rational design process [27].

Understand the Program's Task. Compute the probability that a particular pane of (special) glass will break if an explosive is detonated at a given distance. This could be in the context of the glass facade for a building.

Is it well understood? The details are extensively documented in [3, 4, 7].

Record Base Domain Knowledge. A recurring idea is the different types of `Glass`:

Concept	Term (Name)	Abbrev.	Domain
`fullyT`	Fully Tempered	FT	`[Glass]`
`heatS`	Heat Strengthened	HS	`[Glass]`
`iGlass`	Insulating Glass	IG	`[Glass]`
`lGlass`	Laminated Glass	LG	`[Glass]`
`glassTypeFac`	Glass Type Factor	GTF	`[Glass]`

The "Risk of Failure" is definable:

Label	Risk of Failure
Symbol	B
Units	Unitless
Equation	$B = \frac{k}{(ab)^{m-1}}(Eh^2)^m LDFe^J$
Description	B is the Risk of Failure (Unitless) k is the surface flaw parameter ($\frac{m^{12}}{N^7}$) a & b are the plate length & width (m) m is the surface flaw parameter ($\frac{m^{12}}{N^7}$) E is the modulus of elasticity of glass (Pa) h is the minimum thickness (m) LDF is the load duration factor (Unitless) J is the stress distribution factor (Unitless)
Source	[3], [7]

Some concepts, such as those of *explosion*, *glass slab*, and *degree* do not need to be defined mathematically – an English description is sufficient.

The descriptions in GlassBR are produced using an experimental language using specialized markup for describing relations between knowledge. For example, the goal of GlassBR ("Predict-Glass-Withstands-Explosion") is to "Analyze and predict whether the *glass slab* under consideration will be able to withstand the **explosion** of a certain **degree** that is calculated based on *user input*.", where italicized names are "named ideas", and bold faced names are "concept chunks" (named ideas with a domain of related ideas). We call this goal a "concept instance" (a concept chunk applied in some way). This language lets us perform various static analyses on our artifacts.

This doesn't build a complete ontology of concepts, as we have not found that to be necessary to generate our artifacts. In other words, we define a *good enough* fraction of the domain ontology.

The most important results of this phase are descriptions of *theories* that link all the important concepts together. For example, the description of all the elements that comprise Newton's Law $F = ma$, i.e., what is a force, a mass, acceleration (and how is it related to velocity, position and time), what are the units involved, etc.

Define the characteristics of a good solution. For example, one of our outputs is a probability, which means that the output should be checked to be between 0 and 1. This can result in assertion code in the end program, or tests, or both. We do not yet support full *properties* [9], but that is indeed the logical next step.

Record basic examples. For the purposes of testing, it is always good to have very simple examples, especially ones where the correct answer is known a priori, even though the simple examples are considered "toy problems". They provide a useful extra check that the narrative is coherent with our expectations.

Specialization of theories. In general, the theories involved will be much more general than what is needed in any given example. For example, Newton's Laws are encoded in their vector form in n dimensions, and thus specialization is necessary.

In the GlassBR example, the thickness parameter is not free to vary, but must take one of a specific set of values. The rationale for this specialization is that manufacturers have standardized the glass thickness they will provide. (This rationale is also something we capture.)

Most research software examples involve significant specialization, such as converting partial differential equations to ordinary, elimination of variables, use of closed forms instead of implicit equations, and so on. Often specialization, driven by underlying assumptions, enable further specializations, so that this step is really one of *iterative refinement*.

Create a coherent narrative. Given the outputs we wish to produce, such as the probability that a glass slab will withstand an explosion, we need to ensure that we can go from all the given inputs to the desired output, by stringing together various definitions given in a computational manner. In other words, we need to ensure that there exists a deterministic path from the inputs we are given, through the equations we have, to all of the outputs we've declared we are interested in.

For this example, the computations are all quite simple. The reasons why GlassBR's few lines of code are correct is what is interesting, and that description, which is part of the specification, spans a few pages. That information is generic about the properties of glass and the effects of explosions, and could be reused in other applications. In general, research software tends to involve solving ordinary differential equations, computing a solution to an optimization problem, etc. Again the reason why a particular differential equation is an adequate model of the problem at hand is part of the important information we wish to capture.

In other words, the main narrative of a piece of software is its high level design. What are the techniques used to derive the outputs from the inputs, obtained in a reasoned manner by successive refinements of the underlying theory into an actual program. While the code itself is akin to the actual words used to tell a story, the narrative is the story, its themes and topics, and so on.

Make code level choices. From a *deterministic model of the solution*, it should be possible to output code in a programming language. To get there, we still need to make a series of choices.

Drasil lets you choose output programming language(s) (see Figure 2), but also how "modular" the generated code is, whether we want programs or libraries, the level of logging and comments, etc. Here we show the actual code we use for this, as it is reasonably direct.

```
code :: CodeSpec
code = codeSpec fullSI choices allMods

choices :: Choices
choices = defaultChoices {
 lang = [Python, Cpp, CSharp, Java, Swift],
 modularity = Modular Separated,
 impType = Program, logFile = "log.txt",
 logging = [LogVar, LogFunc],
 comments = [CommentFunc, CommentClass, CommentMod],
 doxVerbosity = Quiet,
 dates = Hide,
 onSfwrConstraint = Exception, onPhysConstraint = Exception,
 inputStructure = Bundled,
 constStructure = Inline, constRepr = Const
}
```

Figure 2 Code level choices for GlassBR (corresponds to much of the material in the bold box in top right of Figure 1).

Create recipes to generate artifacts. The information collected in the previous steps form the core of the various software artifacts normally written. To perform this assembly, we write programs that we dub *recipes*. We have DSLs for creating requirements specifications [31], code [19], dependency diagrams, Makefiles, READMEs and a log of all choices used.

Finally, we can put the contents created via the previous steps together and **generate everything**, here the SRS, the code, traceability graphs and a log of the design choices.

3 An Idealized Process

The above used what we refer to as an *idealized process* for the development of software that is both well-understood and long-lived. Drasil is meant to facilitate this process. In other words, as we ourselves better understand this idealized process, we modify Drasil to follow suit, rather than the other way around. We can summarize the process as follows.

1. Have a task to achieve where *software* can play a central part in the solution.
2. Verify that the underlying problem domain is *well understood*.
3. Describe the problem:
   a. Find the base knowledge (theories) in the pre-existing library or, failing that, write it if it does not yet exist, for instance the naturally occurring known quantities and associated constraints.
   b. Describe the characteristics of a good solution.
   c. Come up with basic examples (to test correctness, intuitions, etc).
4. Describe, by successive refinement transformations, how the problem description can be turned into a deterministic[2] input-output process.
   a. Some refinements will involve *specialization* (e.g., from n-dimensional to 2-dimensional, assuming no friction, etc). These *choices* and their *rationale* need to be documented, as a crucial part of the solution. Whether these choices are (un)likely to change in the future should also be recorded.
   b. Choices tend to be dependent, and thus (partially) ordered. *Decisions* frequently enable or reveal downstream choices.
5. Assemble the ingredients into a coherent narrative.
6. Describe how the process from step 5 can be turned into code. Many choices can occur here as well.
7. Turn the steps (i.e., from items 4 and 6) into a *recipe*, aka program, that weaves together all the information into a variety of artifacts (documentation, code, build scripts, test cases, etc). These can be read, or executed, or ... as appropriate.

The fundamental reason for focusing on *well-understood* software is to make the various steps in this process feasible. Another enabler is a *suitable* knowledge encoding. Rather than define this a priori, we have used a *bottom up* process to capture “good enough” ontologies to get the job done.

What is missing in the above description is an explicit *information architecture* of each of the necessary artifact. In other words, what information is necessary to enable the mechanized generation of each artifact? It turns out that many of them are quite straightforward.

Note that in many software projects, steps 1 and 3 are skipped; this is part of the **tacit knowledge** of a lot of software. Our process requires that we make this knowledge explicit, a fundamental step in *Knowledge Management* [10].

[2] A current meta-design choice.

4 Related Work

As one referee put it, our work is “old fashioned”, in that its goals indeed repeat that of work from decades ago. What we see, however, is that only a small slice of those goals has made it into today’s practices. Thus, we think it is worthwhile to revisit these goals.

Draco [21, 22], one of the oldest such projects, is also one of the closest to Drasil, goal-wise. Many of its ideas have percolated through the literature on DSLs and product lines (see the wonderful textbook [2] for a modern exposition), and we used them without a proper appreciation for their source. While DMS [5, 6], and GLisp [23, 25] are different in their aims, they also contain some of the same ideas (knowledge capture, using various ontologies, refinement and reuse). The examples that drove GLisp (see [24, 16]) are very close to many of ours, namely applications of physics. SpecWare [32, 30] goes much further along the formality spectrum, incorporating correctness proofs as part of its refinement process, as well as being based on a solid theoretical framework.

Unfortunately a lot of the above work was very code-centric. None of the systems generate all software artifacts, especially not documentation. Furthermore, quite a lot of the work uses fairly coarse-grained chunks of information (“components”) which, in our experience, reduces the potential for reuse. Many of the projects were extremely ambitious regarding what aspects of software production they aimed to automate. We do not aim to automate any activity that can reasonably be called *design*.

We have borrowed ideas from product families [28] and software product lines [2]. We have also borrowed ideas from literate programming [15], namely the idea of *chunks* of knowledge that can be written in a different order than what the underlying programming language may want, as well as the idea of tightly coupling explanation with its language encoding. The weaving of multiple languages together, as available via org-mode’s babel-mode [29] to achieve “reproducible research” is a more modern take on literate software development. Jupyter notebooks [14] is a GUI-driven approach that is more beginner-friendly but has scaling drawbacks. We have explicitly decided that it is too early to provide this kind of GUI for Drasil.

Thinking of doing large scale work has been strongly influenced by Eelco Visser’s grand projects. His work on the Spoofax language workbench [13] made it clear that many different artifacts can be generated. He was braver than us: his WebDSL [34], at the heart of researchr [33], is indeed expected to be long lived, but certainly was not created for a well-understood domain! The domain analysis of web applications is a non-trivial contribution of WebDSL, and may well have moved that domain into the “well understood” column. Whether it is product lines or program families, weaving them from DSLs is also something he preached [35]. We never got a chance to present Drasil to Eelco, but we hope that it would have resonated with him.

5 Concluding Remarks

Dines [8] has already remarked that for most software, the *domain knowledge* is the slowest moving part of the requirements. We add to this the ideas that for certain kinds of software, there is significant *knowledge duplication* amongst the artifacts, much of which is traceable to domain knowledge. Thus, in well-understood domains, it should be feasible to record the domain knowledge “once”, and then write recipes to generate instances based on refinements. For long-lived software, this kind of up-front investment should be worthwhile.

In other words, if we capture the fundamental domain knowledge of specific domains (such as mechanics of rigid body motion, dynamics of soil, trains, etc), most later development time can be spent on the *specifics* of the requirements of a specialized application that may well contain novel ideas at the refinement or recipe level.

In Drasil, as a side effect of organizing things as we have, we obtain traceability and consistency, by construction. Tracking where we use each concept, i.e. traceability, is illustrated in Figure 1. We obtain consistency in our documentation by generating items such as the table of symbols from those symbols actually used in the document, and whose definition is automatically extracted from the base knowledge.

There are further ideas that co-exist smoothly with our framework, most notably software families, software product lines and correctness, particularly correctness of documentation. As we generate the documentation in tandem with, and using the same information as, the source code, these will necessarily be synchronized. Errors will co-occur in both. This is a feature, as they are more likely to be caught that way.

We've also noticed that we can more easily experiment with "what if" scenarios, which make it easy to understand the ramifications of proposed changes.

We expect to use formal ontologies as we implement more coherence checks on the domain knowledge itself. For example, it should not be possible to associate a *weight* attribute to the concept *program name*, but only to concepts that are somehow "physical". Perhaps a large ontology in the style of Cyc [17] would help. We also hope to develop a language of design to embody patterns of choices. Many projects already mentioned have such a language that we are eager to borrow from.

More people should resurrect older ideas that were too ambitious for their time, and apply modern understanding and tooling to them.

References

1 Christopher Alexander. *A pattern language: towns, buildings, construction.* Oxford university press, 1977.

2 Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines.* Springer, 2016.

3 ASTM. Standard practice for determining load resistance of glass in buildings, 2009.

4 ASTM. Standard practice for specifying an equivalent 3-second duration design loading for blast resistant glazing fabricated with laminated glass, 2015.

5 Ira D Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992.

6 Ira D Baxter. Dms: Program transformations for practical scalable software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 48–51, 2002.

7 W. Lynn Beason, Terry L. Kohutek, and Joseph M. Bracci. Basis for ASTME E 1300 annealed glass thickness selection charts. *Journal of Structural Engineering*, 124(2):215–221, February 1998.

8 Dines Bjørner. *Domain Science and Engineering.* Monographs in Theoretical Computer Science. An EATCS Series. Springer International Publishing, New York, 2021. doi:10.1007/978-3-030-73484-8.

9 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

10 Kimiz Dalkir. *Knowledge Management in Theory and Practice.* The MIT Press, Cambridge, Massachusetts, USA, 2nd edition, 2011.

11 William M. Farmer. Biform theories in chiron. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, pages 66–79, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

12 Madec Gurvan, Romain Bourdallé-Badie, Jérôme Chanut, Emanuela Clementi, Andrew Coward, Christian Ethé, Doroteaciro Iovino, Dan Lea, Claire Lévy, Tomas Lovato, Nicolas Martin, Sébastien Masson, Silvia Mocavero, Clément Rousset, Dave Storkey, Simon Müeller, George Nurser, Mike Bell, Guillaume Samson, Pierre Mathiot, Francesca Mele, and Aimie Moulin. Nemo ocean engine, March 2022. doi:10.5281/zenodo.6334656.

13 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *ACM SIGPLAN Notices*, 45(10):444–463, October 2010. OOPSLA 2010.

14 Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

15 Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. doi:10.1093/comjnl/27.2.97.

16 Hyung Joon Kook and Gordon S. Novak. Representation of models for expert problem solving in physics. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):48–54, 1991.

17 Douglas B Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

18 Robyn Lutz, David Weiss, Sandeep Krishnan, and Jingwei Yang. Software product line engineering for long-lived, sustainable systems. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 430–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

19 Brooks MacLachlan, Jacques Carette, and Spencer S. Smith. Gool: Generic object-oriented language. In *Proceedings of 2020 SIGPLAM Workskop on Partial Evaluation and Program Manipulation (PEPM 2020)*. ACM, 2020. doi:10.1145/3372884.3373159.

20 G. Madec, P. Delecluse, M. Imbard, and C. Levy. Opa 8 ocean general circulation model - reference manual. Technical report, LODYC/IPSL Note 11, 1998.

21 James M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–574, 1984. doi:10.1109/TSE.1984.5010280.

22 James M Neighbors. Draco: A method for engineering reusable software systems. *Software reusability*, 1:295–319, 1989.

23 Gordon S Novak. Glisp: A lisp-based programming system with data abstraction. *AI Magazine*, 4(3):37–37, 1983.

24 Gordon S Novak. Generating programs from connections of physical models. In *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*, pages 224–230. IEEE, 1994.

25 Gordon S. Novak. Creation of views for reuse of software with different data representations. *IEEE Trans. Software Eng.*, 21(12):993–1005, 1995. doi:10.1109/32.489074.

26 David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, New York, 2011.

27 David Lorge Parnas and Paul C Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

28 D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976. doi:10.1109/TSE.1976.233797.

29 Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46:1–24, 2012.

30 Douglas Smith. *Mechanizing the Development of Software*, pages 251–292. IOS Press, August 1999.

31 W. Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*, 13(1):83–107, February 2007. doi:10.1007/s11155-006-9020-7.

32 Y. V. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Mathematics of Program Construction*, pages 399–422, 1995.

33 Elmer van Chastelet, Eelco Visser, and Craig Anslow. Conf.researchr.org: Towards a domain-specific content management system for managing large conference websites. In Jonathan Aldrich and Patrick Eugster, editors, *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications (SPLASH): Software for Humanity*, pages 50–51. ACM, 2015.

34 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, LNCS 5235. Springer, July 2008. GTTSE 2007.

35 Markus Voelter and Eelco Visser. Product line engineering using domain-specific languages. In *2011 15th International Software Product Line Conference*, pages 70–79. IEEE, 2011.

Stack Graphs

Name Resolution at Scale

Douglas A. Creager
GitHub, Southborough, MA, USA

Hendrik van Antwerpen
GitHub, Amsterdam, The Netherlands

Abstract

We present *stack graphs*, an extension of Visser et al.'s scope graphs framework. Stack graphs power Precise Code Navigation at GitHub, allowing users to navigate name binding references both within and across repositories. Like scope graphs, stack graphs encode the name binding information about a program in a graph structure, in which paths represent valid name bindings. Resolving a reference to its definition is then implemented with a simple path-finding search.

GitHub hosts millions of repositories, containing petabytes of total code, implemented in hundreds of different programming languages, and receiving thousands of pushes per minute. To support this scale, we ensure that the graph construction and path-finding judgments are *file-incremental*: for each source file, we create an isolated subgraph without any knowledge of, or visibility into, any other file in the program. This lets us eliminate the storage and compute costs of reanalyzing file versions that we have already seen. Since most commits change a small fraction of the files in a repository, this greatly amortizes the operational costs of indexing large, frequently changed repositories over time. To handle type-directed name lookups (which require "pausing" the current lookup to resolve another name), our name resolution algorithm maintains a stack of the currently paused (but still pending) lookups. Stack graphs can be constructed via a purely syntactic analysis of the program's source code, using a new declarative *graph construction language*. This means that we can extract name binding information for every repository without any per-package configuration, and without having to invoke an arbitrary, untrusted, package-specific build process.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Software libraries and repositories

Keywords and phrases Scope graphs, name binding, code navigation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.8

Supplementary Material *Software*: `https://github.com/github/stack-graphs/` [3]
archived at `swh:1:rel:06b63a06504d38c7a0e9ee03b1d21f6f3adc045a`
Software: `https://github.com/tree-sitter/tree-sitter-graph/` [16]
archived at `swh:1:rel:6877dac0c0f228f6450a2a2a6fdae178947e60ef`

1 Introduction

Code editors have long provided productivity features like *code navigation*, which let the user see and navigate the structural relationships in their code – specifically which entities the names in their code refer to. These features are equally useful on a software forge like GitHub. However, large software forges must support a completely different magnitude of scale, along a number of axes: the total volume of stored code; the number of changes received over time; the need to query historic versions of the code; the number of users making queries simultaneously; and the number of programming languages to support.

We have developed *stack graphs* to address these constraints. Stack graphs build on Visser et al.'s *scope graphs* framework [17], which use a graphical notation to encode the name binding rules for a programming language. Like scope graphs, each name binding in a program is represented by a path in the corresponding graph. Unlike scope graphs, our path-finding algorithm maintains a *stack* of pending queries (hence the name), allowing a

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 8; pp. 8:1–8:12
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

name binding to depend on the results of other, intermediate name bindings. Stack graphs are *file-incremental*, meaning that each file in the source program is represented by a disjoint subgraph, with no edges crossing between them. (Special *virtual edges* are created at query time to cross between files.) We can construct each file's subgraph in isolation, without inspecting any of the other files in the program. This lets us easily detect and reuse results for file versions that we have already seen and analyzed. Since most commits in a project's history change a small fraction of files in the project, this greatly reduces the computational and storage costs of providing this service.

This paper is organized as follows. In §2, we provide an overview of scope graphs and the limitations that led to the development of stack graphs. In §3 and §4, we describe the stack graphs formalism in detail. In §5, we discuss related work. In §6, we conclude.

2 Background and historical perspective

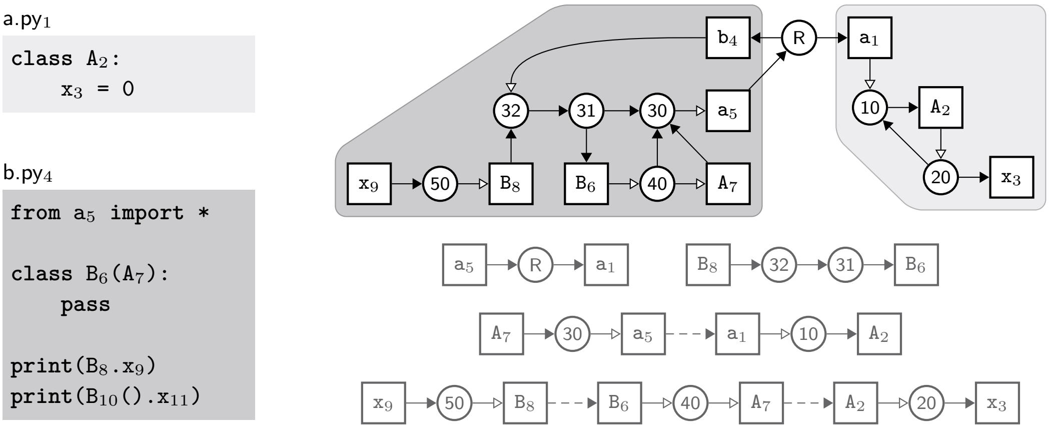
We first started investigating how best to add code navigation capabilities to GitHub in 2017. Code navigation is widely available in local editors, so an appealing potential approach would be to adopt whatever technology powers the local editor solution, porting it as necessary to run in our production web server environment.

However, there are clear differences between the code navigation experience in a local editor compared with a software forge. Unlike the local editor, where there is a single user viewing and interacting with a single project (and primarily a single current version of the project's code), a software forge must support an unbounded number of users simultaneously viewing any historic version of any of the code hosted on the forge. Moreover, we must support users who are *exploring* and *browsing* the code, in addition to those who are *authoring* and *maintaining* it. These explorers often view code months or years after it has been written. This means that we must either keep our code navigation data available, and readily queryable, indefinitely; or ensure that we can (quickly) regenerate that data on demand in response to a future user query. This leads to a clear distinction between *index time*, when a user "pushes" a new snapshot of a program or library to the forge, and *query time*, when a (likely different) user wants to view or explore that snapshot at some point in the future.

We have ambitious but conflicting latency goals for the index and query phases. Our most important goal is to minimize the delay between a user performing a code navigation query and us displaying the results of that query in their browser [9]. To achieve this, we must spend some time precalculating information at index time. That said, we do not want to perform *too much* precalculation work, so that code navigation is available quickly after each push,[1] and so we do not waste time precalculating code navigation data for a repository or commit that will never be viewed.

Our scale also presents unique challenges. The most obvious is the volume of code: GitHub holds petabytes of code history, and receives thousands of new code snapshots each minute. It is imperative that our framework be *incremental*, skipping the costs of reanalyzing code that we have already seen. Luckily, much of each project's history is redundant, with most commits changing only a small fraction of files. As such, we particularly prefer *file-incremental* analyses, where we can analyze each source file at index time in isolation, without inspecting (or having access to) any of the other files in the project. The Merkle tree [7] data

[1] For querying, our goal is to show results in under 100 ms. For indexing, our less precise goal is that code navigation should be available before the user can Alt-Tab from their git client over to the browser.



a.py$_1$

```
class A2:
    x3 = 0
```

b.py$_4$

```
from a5 import *

class B6(A7):
    pass

print(B8.x9)
print(B10().x11)
```

Figure 1 A sample program and its Néron scope graph.

model used by git provides a unique *blob identifier* for each file version, which depends only on the file's content. Because a file's identifier is available before analysis starts, we can skip the storage *and computation* costs of redundant file-incremental work.

Lastly, GitHub hosts code written in a staggering number of programming languages.[2] We want code navigation to work for all of them. It will never be cost-effective for GitHub engineers to implement support for the long tail of less popular languages. But there must be a path for *someone* – whether a GitHub engineer or a member of an external language community – to add support for every programming language that exists and is in use.

As we elaborate in §5, existing local editor code navigation solutions do not satisfy these constraints. While researching whether existing academic work could help tackle this problem, we discovered the *scope graphs* framework [17], which introduced a novel and intuitive approach for encoding the name binding semantics of a programming language in a graph structure.[3] This seemed likely to satisfy our language support goals: all language-specific logic would be isolated to the graph construction process, and all querying would happen via a single language-independent algorithm that operated on that graph structure. However, we quickly discovered that scope graphs, as originally described, would not meet our scale requirements. In the rest of this section, we will explore why, using a simple Python program as a running example.

2.1 Néron scope graphs

Figure 1 shows our example Python program, which consists of two files. The first defines a class $\mathtt{A}_2$ containing a single class field $\mathtt{x}_3$.[4] The second file imports all of the names from $\mathtt{a}_5$, defines a subclass $\mathtt{B}_6$, and then prints its inherited field twice: by accessing it first as a class member and then as an instance member.

[2] As of this writing, Linguist [4], our open-source language detection library, includes over 500 distinct languages and sublanguages in its ruleset.

[3] Note that our initial discovery and investigation of the scope graphs framework predates van Antwerpen joining GitHub.

[4] Following the scope graph literature, we add a unique numeric subscript to each identifier so that we can easily distinguish different occurrences of the same identifier in the program source. These subscripts are not part of the actual identifiers seen by the Python interpreter.

We also show the scope graph that we would construct for this program according to the judgments defined by Néron et al. [10]. Circular nodes represent *scopes*, which are "minimal program regions that behave uniformly with respect to name resolution."[5] Rectangular nodes represent definitions and references, with edges connecting a scope to each of the symbols defined in that scope, and connecting each reference to the scope in which that symbol should be resolved. Scope Ⓡ is a *root node*, representing the global namespace that all Python modules belong to, while $\mathtt{a}_1$ and $\mathtt{b}_4$ are the definitions of the modules defined in the two files.[6] Scope ⑳ contains the class members of class `A`, while the edge from ⑳ to $\mathtt{x}_3$ indicates that `x` is one of those class members. Scope ㊿ represents the names that are available via the member access operator in the first `print` statement, while the edge from $\mathtt{x}_9$ to ㊿ indicates that the reference must be resolved relative to those names.

Edges between scopes represent nesting. For instance, the top-level definitions in a Python module are evaluated sequentially, and each name is only visible after its definition has been evaluated. This is modeled by the edges connecting scopes ㉚, ㉛, and ㉜, which represent the names visible immediately after the `import`, `class`, and first `print` statements, respectively. Scope ㊵ contains the class members of class `B`; the edge between it and ㉚ models how references within the class body can refer to definitions in the containing lexical scope. Open-arrow edges represent "imports," which make the contents of a named scope available elsewhere. For instance, the edge between ㊵ and $\mathtt{A}_7$ specifies that `B` inherits all of the class members of `A`, while the edge between $\mathtt{A}_2$ and ⑳ specifies that scope ⑳ contains the class members of class `A`.

We also show the name binding path that correctly resolves the reference $\mathtt{x}_9$ to its definition $\mathtt{x}_3$. This requires resolving several intermediate references via import edges: resolving $\mathtt{B}_8$ to the definition of class `B`; $\mathtt{A}_7$ to its superclass; and $\mathtt{a}_5$ to the module defining that superclass.

Importantly, the Néron scope graph can be divided into disjoint subgraphs for each file, indicated by shading. Apart from the root node (which is a singleton node shared across all files), every node belongs to some file's subgraph. Every edge connects two nodes that belong to the same file, or connects a node to the shared root node. At query time, we must consider the scope graph as a whole, since name binding paths can cross from one subgraph to another. However, we can *construct* each file's subgraph at index time without having to consider the content of any other file in the program. Néron scope graph construction is therefore file-incremental in exactly the way that we need.

2.2 Van Antwerpen scope graphs

Unfortunately, the Néron scope graph does not allow us to resolve $\mathtt{x}_{11}$, since it depends on *type-dependent name resolution.* To resolve $\mathtt{x}_{11}$, we must know the return type that we get from invoking $\mathtt{B}_{10}$, so that we can resolve the reference in the return type's scope. The return type will depend on what kind of entity $\mathtt{B}_{10}$ resolves to, which we do not know *a priori.* In this particular example, it resolves to a class definition, and so the return type is an instance of that class. But it could resolve to a function (or any other "callable"), with an arbitrary return type defined by the function body.

[5] We have been careful to label scope nodes consistently across all of our examples: scopes with the same number in each figure represent the same regions of the example source program. In the grand tradition of a BASIC programmer choosing line numbers, we have left "gaps" in the sequence of assigned scope numbers so that related scopes can have numbers close to each other, while leaving room to add additional scopes in later examples.

[6] Note that in Python, the name of a module does not appear explicitly in the source code, and is instead inferred from the name of the file defining it.

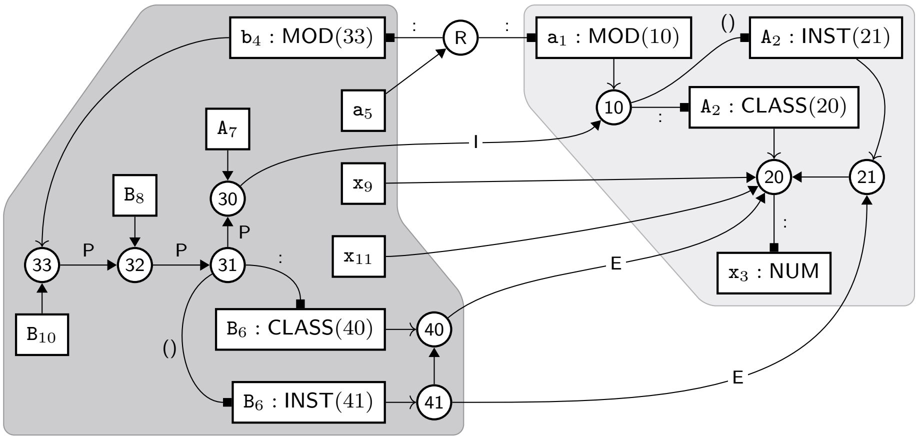


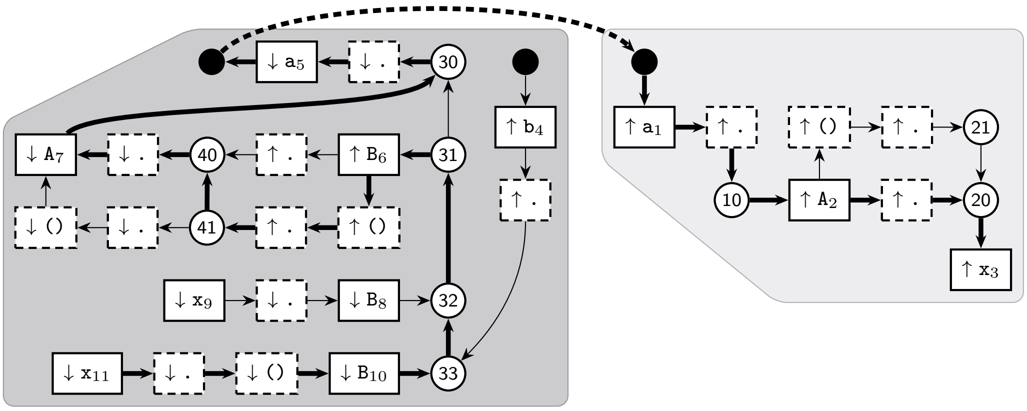
Figure 2 The sample program's van Antwerpen scope graph.

Van Antwerpen et al. [15, 13] extend the scope graph judgments to support these kinds of lookups. Figure 2 shows the scope graph for our example program using these modified judgments. We use square-cap edges to connect each scope with the definitions in that scope. These edges are labeled to indicate the kind of definition: a `:` edge specifies the type of a symbol defined in the scope, while a `()` edge specifies the return type of a callable symbol defined in the scope. We need edges with both labels for Python classes, since they can both be referred to by name (yielding the class itself) and called (yielding an instance of the class). Similarly, we have two associated scopes for each class: one (⑳ and ㊵) for the class members, and one (㉑ and ㊶) for the instance members. Each pair of scopes is connected by an edge, modeling how class members are also available as instance members.

There are paths in the van Antwerpen scope graph that resolve both x_9 and x_{11} to the definition x_3. However, the paths encoding these name bindings (x_9 → ⑳ → x_3 and x_{11} → ⑳ → x_3) are shorter than we might expect, since they do not directly encode all of the intermediate lookups that are needed.

The intermediate lookups do happen, but at a different time. In the Néron scope graph, there are import edges connecting ㉚ to a_5 and a_1 to ⑩, which we follow lazily at query time. In the van Antwerpen scope graph, we perform these intermediate lookups eagerly at graph construction time. When we encounter the `import` statement in the source, we immediately resolve the a_5 reference. Doing so takes us to a_1 and its connected scope ⑩, as in the Néron scope graph. However, we then persist this lookup result into the graph structure, as an I edge directly connecting ㉚ to ⑩. We perform (and persist) similar construction-time lookups of B's superclass reference A_7, and of the type references B_8 and B_{10}.

While we have gained the ability to perform type-dependent lookups, we have lost file incrementality. The eager intermediate lookups produce edges that cross file boundaries, violating the disjointedness property of Néron scope graphs. This violation is not superficial; it highlights that each intermediate lookup could depend on *any other* file in the program. We cannot analyze each file purely in isolation, since we cannot know in advance which other files we might have to inspect. Worse, future updates to other files might invalidate the subgraph that we create. In the pathological case, a change to one file might require us to regenerate the scope graph for the *entire program*.



$\downarrow \mathtt{x}_{11} \rightsquigarrow \downarrow \mathtt{x}_{11}$	$\langle\mathtt{x}\rangle$		$\rightsquigarrow \uparrow\ .$	$\langle\mathtt{x}\rangle$		$\rightsquigarrow \uparrow \mathtt{a}_1$	$\langle\mathtt{.A.x}\rangle$	
$\rightsquigarrow \downarrow\ .$	$\langle\mathtt{.x}\rangle$		$\rightsquigarrow$ ㊶	$\langle\mathtt{x}\rangle$	(2)	$\rightsquigarrow \uparrow\ .$	$\langle\mathtt{A.x}\rangle$	
$\rightsquigarrow \downarrow\ ()$	$\langle\mathtt{().x}\rangle$		$\rightsquigarrow$ ㊵	$\langle\mathtt{x}\rangle$		$\rightsquigarrow$ ⑩	$\langle\mathtt{A.x}\rangle$	(5)
$\rightsquigarrow \downarrow \mathtt{B}_{10}$	$\langle\mathtt{B().x}\rangle$		$\rightsquigarrow \downarrow\ .$	$\langle\mathtt{.x}\rangle$		$\rightsquigarrow \uparrow \mathtt{A}_2$	$\langle\mathtt{.x}\rangle$	
$\rightsquigarrow$ ㉝	$\langle\mathtt{B().x}\rangle$	(1)	$\rightsquigarrow \downarrow \mathtt{A}_7$	$\langle\mathtt{A.x}\rangle$		$\rightsquigarrow \uparrow\ .$	$\langle\mathtt{x}\rangle$	
$\rightsquigarrow$ ㉜	$\langle\mathtt{B().x}\rangle$		$\rightsquigarrow$ ㉚	$\langle\mathtt{A.x}\rangle$	(3)	$\rightsquigarrow$ ⑳	$\langle\mathtt{x}\rangle$	(6)
$\rightsquigarrow$ ㉛	$\langle\mathtt{B().x}\rangle$		$\rightsquigarrow \downarrow\ .$	$\langle\mathtt{.A.x}\rangle$		$\rightsquigarrow \uparrow \mathtt{x}_3$	◇	
$\rightsquigarrow \uparrow \mathtt{B}_6$	$\langle\mathtt{().x}\rangle$		$\rightsquigarrow \downarrow \mathtt{a}_5$	$\langle\mathtt{a.A.x}\rangle$				
$\rightsquigarrow \uparrow\ ()$	$\langle\mathtt{.x}\rangle$		$\rightsquigarrow$ ●	$\langle\mathtt{a.A.x}\rangle$	(4)			

Figure 3 The sample program's stack graph.

3 Stack graphs

Stack graphs support the advanced type-dependent lookups of van Antwerpen scope graphs. They also retain the file incrementality of Néron scope graphs, by performing intermediate lookups lazily at query time. Our key insight is to maintain an explicit *stack* of the currently pending intermediate lookups during the name resolution algorithm.

Figure 3 shows the stack graph for our example program. There is no longer a shared singleton root node; the graph can contain multiple root nodes, which are indicated by filled circles. Definition and reference nodes have a solid border. Nodes with a dashed border are *push* and *pop* nodes; we will see below how they enable type-dependent lookups. We add arrows to clearly distinguish definition and pop nodes (↑) from reference and push nodes (↓).

We also highlight a name binding path that resolves $\mathtt{x}_{11}$ to $\mathtt{x}_3$, and trace the discovery of that path. We start with an "empty" path from $\mathtt{x}_{11}$ to itself. With each step, we append an edge to the path, and show the path's current frontier and the stack of currently pending lookups. This stack contains both program identifiers (e.g. `x`) indicating names that we need to resolve, and operators (e.g. `.`) indicating how the resolved definitions will be used. Whenever we encounter a reference or push node, we prepend the node's symbol to the stack. Whenever we encounter a definition or pop node, we verify that the stack starts with the node's symbol, and remove it from the stack. The final name binding path ends at a definition node and has an empty stack, indicating that there are no more pending lookups, and the name binding path is complete.

Several steps of interest are highlighted. At step (1) we have seeded the stack with a representation of the full expression being resolved. The top of the stack specifies that the first intermediate lookup that we must perform is to resolve $\mathtt{B}_{10}$. The path's frontier indicates that this initial lookup is to be performed in the context of scope ㉝, which represents the names that are visible at the end of the module.

$$\begin{array}{rlrl}
\textit{stack graph} & G & \textit{node} & \mathcal{N}^i_G \; ::= \; \bullet \mid \bigcirc \mid \downarrow x \mid \uparrow x \\
\textit{source program} & P & \textit{edge} & \mathcal{E}_G \; \ni \; i \to i' \\
\textit{symbol} & x & \textit{symbol stack} & \hat{x} \; ::= \; \diamond \mid x \cdot \hat{x} \\
\textit{node identifier} & i & \textit{path} & p \; ::= \; i \rightsquigarrow i' \; \{\hat{x}\} \\
\textit{source file} & \mathcal{F}^i_G & & \\
 & & & i \to i' \in \mathcal{E}_G \Rightarrow \mathcal{F}^i_G = \mathcal{F}^{i'}_G
\end{array}$$

Figure 4 Stack graph structure and paths.

At step (2) we have resolved $\texttt{B}_{10}$, and have just popped off the `()` and `.` operator symbols to determine what occurs when we invoke its definition and perform member access on the result. The stack still contains `x`, which will be resolved next. The path's frontier is ㊶, indicating that $\texttt{x}_{11}$ will be resolved as an instance member of `B`. The very next edge takes us to ㊵, which allows us to attempt to resolve $\texttt{x}_{11}$ as a class member, remembering that the instance members of a class include all of its class members.

At step (3) we have followed the edges modeling the class inheritance. Doing so pushes additional symbols onto the stack, indicating that we might find `x` as an inherited class member of `A`. Our next step is to resolve $\texttt{A}_7$ in the context of the path's frontier – scope ㉚, which represents the names that are visible immediately before `B`'s class definition.

At step (4) we have followed the edges representing the `import` statement. The stack now describes how *any* symbol that we are currently looking for might be imported from module `a`. The path's frontier is a root node. Root nodes are the only way that a name binding path can cross from one file to another: when we encounter a root node, we can add a *virtual edge* (the dashed edge in Figure 3) to any other root node, in any file.

At step (5) we have resolved $\texttt{a}_5$ to scope ⑩, which represents the definition of the imported module. At step (6), we have resolved $\texttt{A}_7$ to scope ⑳, which contains the class members of class `A`. From there, all that remains is to resolve $\texttt{x}_{11}$ to its definition, $\texttt{x}_3$.

The formal definition of a stack graph is shown in Figure 4. A stack graph G is a representation of a *source program* P, which consists of a set of *source files*. Each source file can be parsed into a set of *syntax nodes*. A subset of those syntax nodes represent *definitions*, and a different (not necessarily disjoint) subset of syntax nodes represent *references*. A *symbol* x is an identifier from the source language, representing (part of) the name of an entity in the program, or an operator symbol like `.` or `()`.

Each node in a stack graph has a unique *node identifier*, and must be one of the following: a *root* node ●, a *scope* node ◯, a *push symbol* node $\downarrow x$, or a *pop symbol* node $\uparrow x$. $\mathcal{N}^i_G$ denotes the node in G with node identifier i. Each node *belongs to* exactly one source file, denoted $\mathcal{F}^i_G$. Some nodes belong to a specific syntax node within that file. A pop symbol node is a *definition node* if it belongs to a syntax node that is a definition. A push symbol node is a *reference* if it belongs to a syntax node that is a reference. $\mathcal{E}_G$ denotes the set of edges in stack graph G. Each edge e in a stack graph is directed, connecting a *source node* i to a *sink node* i'. Edges can only connect nodes that belong to the same file.

The judgment $G \vdash p$ states that p is a valid path in stack graph G. A path consists of a *start node* i, an *end node* i', and a *symbol stack* $\hat{x}$. A path is *complete* if its start node is a reference node, its end node is a definition node, and its symbol stack is empty. Every name binding in a source program P is represented by a complete path in the corresponding stack graph G.

Paths are constructed according to the rules in Figure 5. An *empty* path is one with no edges. It is created by *lifting* a stack graph node. Push symbol nodes "seed" the path's symbol stack. Pop symbol nodes cannot be lifted into paths. All other nodes result in an

$$\textsc{LiftPush}\quad \frac{\mathcal{N}_G^{i} = \downarrow x}{G \vdash i \rightsquigarrow i \ \{x\}}$$

$$\textsc{LiftNoop}\quad \frac{\mathcal{N}_G^{i} \in \{\bullet, \bigcirc\}}{G \vdash i \rightsquigarrow i \ \{\diamond\}}$$

$$\textsc{Noop}\quad \frac{G \vdash i_0 \rightsquigarrow i_1 \ \{\hat{x}\} \qquad i_1 \to i_2 \in \mathcal{E}_G \qquad \mathcal{N}_G^{i_2} \in \{\bullet, \bigcirc\}}{G \vdash i_0 \rightsquigarrow i_2 \ \{\hat{x}\}}$$

$$\textsc{Push}\quad \frac{G \vdash i_0 \rightsquigarrow i_1 \ \{\hat{x}\} \qquad i_1 \to i_2 \in \mathcal{E}_G \qquad \mathcal{N}_G^{i_2} = \downarrow x}{G \vdash i_0 \rightsquigarrow i_2 \ \{x \cdot \hat{x}\}}$$

$$\textsc{Pop}\quad \frac{G \vdash i_0 \rightsquigarrow i_1 \ \{x \cdot \hat{x}\} \qquad i_1 \to i_2 \in \mathcal{E}_G \qquad \mathcal{N}_G^{i_2} = \uparrow x}{G \vdash i_0 \rightsquigarrow i_2 \ \{\hat{x}\}}$$

$$\textsc{Root}\quad \frac{G \vdash i_0 \rightsquigarrow i_1 \ \{\hat{x}\} \qquad \mathcal{N}_G^{i_1} = \bullet \qquad \mathcal{N}_G^{i_2} = \bullet \qquad i_1 \neq i_2}{G \vdash i_0 \rightsquigarrow i_2 \ \{\hat{x}\}}$$

Figure 5 Constructing paths by lifting nodes and appending edges.

empty path with an empty symbol stack. We can extend a path by *appending* any edge whose start node is the same as the path's end node. The path's symbol stack might change, depending on the edge's end node. Root nodes and scope nodes are "noops," which leave the symbol stack unchanged. Push symbol nodes prepend an element onto the symbol stack. Pop symbol nodes act as a "guard," requiring that the top of the symbol stack match the node's symbol. This symbol is removed from the symbol stack. (This explains why pop symbol nodes cannot be lifted into empty paths, since there is no symbol stack yet to satisfy this constraint.) If a path ends at a root node, we can immediately extend it to any other root node in the stack graph via a *virtual edge*. (This is the only way that paths cross from one file to another.)

Given these path construction judgments, we can implement a *jump to definition* algorithm: given a source program, and a reference in the program, we want to find all of the definitions that the reference resolves to.[7] First, we construct the stack graph for the program. We then perform a breadth-first search of the graph, maintaining a set of *pending* paths, starting with the empty path we get from lifting the reference node. We use a fixed-point loop to find new pending paths by appending (possibly virtual) edges to existing ones. When appending edges, we must satisfy the constraints of the path construction judgments in Figure 5, and be careful to detect cycles. Whenever we encounter a path that is complete, its end node identifies one of the definitions that the reference resolves to.

This algorithm can be divided such that we construct the stack graph at index time, and save a representation of the graph to persistent storage. At query time, we load in the stack graph and perform the path-finding search. This allows us to amortize the cost of constructing the stack graph across multiple queries. Moveover, stack graph construction is file-incremental, since each node and (non-virtual) edge belongs to exactly one file. By structuring the persistent representation of the stack graph so that each file's subgraph can be identified and stored independently, we can track which file versions we have already created subgraphs for. When we receive a new commit for a repository at index time, we only have to parse and generate new subgraphs for the file versions not seen in previously indexed commits.

[7] When compiling or executing a program, the source language's semantics will typically require that each reference resolve to *exactly one* definition. For an exploratory feature like code navigation, we loosen this restriction to allow ambiguous and missing bindings. This lets us present useful information even in the presence of incorrect programs or an incomplete model of the language's semantics.

$$\begin{array}{rrcl} \textit{symbol stack variable} & \psi & & \\ \textit{partial symbol stack} & \Psi & := & \hat{x} \mid \hat{x} \cdot \psi \\ \textit{partial path} & \tilde{p} & := & \{\Psi\}\ i \rightsquigarrow i'\ \{\Psi'\} \end{array}$$

$$[\![i \rightsquigarrow i'\ \{\hat{x}\}]\!] \triangleq \{\diamond\}\ i \rightsquigarrow i'\ \{\hat{x}\}$$

$$\begin{array}{rcccl} \{\diamond\} & \downarrow \mathtt{x}_{11} & \rightsquigarrow & \textcircled{33} & \{\langle \mathtt{B().x} \rangle\} \\ \{\langle \mathtt{B().} \rangle \cdot \psi\} & \textcircled{33} & \rightsquigarrow & \textcircled{41} & \{\psi\} \\ \{\psi\} & \textcircled{41} & \rightsquigarrow & \textcircled{30} & \{\langle \mathtt{A.} \rangle \cdot \psi\} \\ \{\psi\} & \textcircled{30} & \rightsquigarrow & \bullet & \{\langle \mathtt{a.} \rangle \cdot \psi\} \\ \{\langle \mathtt{a.} \rangle \cdot \psi\} & \bullet & \rightsquigarrow & \textcircled{10} & \{\psi\} \\ \{\langle \mathtt{A.} \rangle \cdot \psi\} & \textcircled{10} & \rightsquigarrow & \textcircled{20} & \{\psi\} \\ \{\langle \mathtt{x} \rangle \cdot \psi\} & \textcircled{20} & \rightsquigarrow & \uparrow \mathtt{x}_3 & \{\psi\} \end{array}$$

Figure 6 Partial paths.

4 Optimizing queries using partial paths

The process described in §3 splits work between index time and query time, and ensures that the work we do at index time is file-incremental, with all *non*-file-incremental work happening at query time. Unfortunately, this process performs *too much* work at query time. The path-finding search is not cheap, and as users perform many queries over time, we will duplicate the work of discovering the overlapping parts of the resulting name binding paths.

We would like to shift some of this work back to index time, while ensuring that all index-time work remains file-incremental. To do this, we calculate *partial paths* for each file, which precompute portions of the path-finding search. Because stack graphs have limited places where a path can cross from one file into another, we can calculate all of the possible partial paths that remain within a single file, or which reach an "import/export" point.

Partial paths are defined in Figure 6. A partial path consists of a *start node* i, an *end node* i', and a *precondition* Ψ and *postcondition* Ψ', which are each partial symbol stacks. A *partial symbol stack* is a symbol stack with an optional *symbol stack variable.* (Note that a symbol stack variable can only appear at the *end* of a partial symbol stack.) Every path has an equivalent partial path whose precondition is empty, and whose postcondition is the path's symbol stack.

Partial paths are constructed using *lift* and *append* judgments similar to those for paths. Partial paths can be *concatenated* by unifying the left-hand side's postcondition with the right-hand side's precondition, and substituting any resulting symbol stack variable assignments into the left-hand side's precondition and right-hand side's postcondition.

Figure 6 also shows several example partial paths. Each partial path precomputes a portion of the name binding path that we traced through in Figure 3, starting and/or ending at one of the highlighted steps. These partial paths can be concatenated together, yielding the complete name binding path from $\mathtt{x}_{11}$ to $\mathtt{x}_3$.

Partial paths give us a better balance of work between index time and query time. At index time, we parse each previously unseen source file and produce a stack subgraph for it, as before. We then find all partial paths within the file between certain *endpoint* nodes (root nodes, definition and reference nodes, and certain important scope nodes). Instead of saving the subgraph structure to persistent storage, we save this list of partial paths. At query time, our algorithm has the same overall structure as before. Instead of tracking pending paths and appending compatible edges to them, we track pending partial paths and concatenate them with compatible *partial path extensions.* This process is guaranteed to find all name binding paths that satisfy the path construction judgments from §3.[8] Like our previous algorithm, we amortize the cost of stack graph construction across multiple queries by performing it once at index time. By precalculating partial paths at index time, the new algorithm also amortizes large parts of the path-finding search.

[8] Due to space limitations, we do not provide full definitions of partial path construction or concatenation or a proof of this guarantee. These will appear in a future paper.

5 Discussion

In this section we describe how stack graphs relate to other work in this area. The most obvious comparison is with scope graphs. In §3 we described how stack graphs were inspired by scope graphs, and were developed in an attempt to support type-dependent lookups while retaining file incrementality.

Ours is not the only attempt to add incrementality to an existing program analysis. Other approaches typically focus on what we term *delta incrementality* [5, 6]. Given a full result set, one determines which files each result depends on. When a file is then changed, only a subset of results are invalidated and recomputed.

Zwaan [18] shows how to add delta incrementality to van Antwerpen scope graphs. In our running example, this approach can detect if an edit potentially causes $\mathtt{x}_{11}$ to resolve to some other definition. If so, the entire file is reanalyzed. The updated scope graph still has edges that cross file boundaries, whose sink nodes might have changed due to edits in other files. This means that we must store separate copies of each file's scope subgraph for each commit that the file version appears in. The resulting storage costs would be prohibitive. In contrast, because stack graph content is file-incremental, each file's subgraph can be stored once, in isolation, and reused however many times that file version appears in the project's history. Moreover, we can detect skippable precalculation work early, using only the blob identifiers provided by git.

The problem of language support is not unique to large software forges like GitHub. Local editors have coalesced around the Language Server Protocol (LSP) standard [8], which provides an abstraction barrier between language-specific and editor-specific tooling. Instead of requiring $L \times E$ integrations (one for every combination of language and editor), there are L "server" implementations and E "client" implementations. This greatly reduces the total amount of development work needed to support code navigation "everywhere."

An LSP server typically runs alongside the local editor as an interactive "sidecar" process, answering query requests triggered by user interactions in the editor. This design makes LSP servers unsuitable for large software forges, since it would require maintaining a large enough fleet of LSP servers to handle the maximum expected number of simultaneous user queries. Moreover, we would have to maintain separate fleets of LSP servers for each supported language. The operational burden of maintaining these fleets counteracts the development time saved by reusing existing LSP implementations.

In response to these difficulties, the LSP community developed the Language Server Index Format (LSIF) [2]. This allows LSP servers to run in "batch" mode, producing a list of all name binding resolutions, which can be saved into a simple database table for easy and fast querying at any point in the future. While the LSIF specification is *capable* of storing incremental results from analyzing individual files, no extant LSP *implementations* support that mode of operation. Instead, they are run against an entire project snapshot, typically in the same continuous integration (CI) pipeline used to build and test the project. When a new commit arrives, the entire project needs to be reanalyzed, even if only a small number of files have changed.[9]

[9] LSP servers typically piggy-back on existing compilers and linters, which historically have not worried about incremental operation. Updating the LSP server to be incremental would require retrofitting the existing compiler, which is a more substantial undertaking than writing an incremental LSP server from scratch [11].

LSP and LSIF are the standardization of many previous language-specific frameworks for generating code navigation data at build time. All build-time analyses suffer the same drawbacks. They require the package owner to explicitly specify how to analyze their project,[10] and by running the analysis as part of CI, require the project owner to pay for the compute resources used.

Stack graphs, on the other hand, can be produced via a purely syntactic process, since all name binding logic is encoded in the resulting graph structure. To support this, we have created a new declarative *graph construction language* called tree-sitter-graph [16]. Building on the tree-sitter parsing framework [1], the language expert defines patterns that match against the language's grammar, and which "gadgets" of stack graph nodes and edges should be created for each instance of those patterns in the concrete syntax tree of a source file. These patterns are defined *once* for each language. As a result, stack graph construction requires no configuration by the project owner, and does not need to invoke a project-specific, typically slow, build process.

6 Conclusion

In this paper we have described *stack graphs*, which build upon Visser et al.'s scope graphs framework. Stack graphs can be used to perform type-dependent name resolution, and are file-incremental, which is essential to operate efficiently and cost-effectively at GitHub's scale. Stack graphs have been running in production since November 2021, analyzing every commit to every public and private Python repository hosted on GitHub. The core name resolution algorithm is language-agnostic, and is implemented (and tested, vetted, and deployed to production) *once*. The tree-sitter [1] and tree-sitter-graph [16] libraries, the stack graph algorithms [3, 14], and our initial language-specific stack graph construction rulesets [12] are all open source. This allows other tools to incorporate stack graph code navigation features, and allows external language communities to self-serve support for their language.

References

1 Max Brunsfeld et al. Tree-sitter: An incremental parsing system for programming tools. doi:10.5281/zenodo.4619183.

2 Dirk Bäumer et al. *Language Server Index Format specification, 0.4.0*, 2019. URL: https://github.com/microsoft/language-server-protocol/blob/main/indexFormat/specification.md.

3 Douglas A. Creager and Hendrik van Antwerpen. stack-graphs library, version 0.10.2, January 2023. doi:10.5281/zenodo.7520627.

4 GitHub. Linguist: The language savant. URL: https://github.com/github/linguist/.

5 Sven Kloppenburg. *Incrementalization of Analyses for Next Generation IDEs*. PhD thesis, Technische Universität, Darmstadt, November 2009. URL: http://tuprints.ulb.tu-darmstadt.de/1960/.

6 Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. HyperAST: Enabling efficient analysis of software histories at scale. In *The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, October 2022.

7 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. doi:10.1007/3-540-48184-2_32.

[10] Modern languages tend to provide official package managers and build tools, and often admit good heuristics for how to build a typical project. But this is not generally true for all languages.

8 Microsoft. *Language Server Protocol specification, 3.17*, May 2022. URL: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/.

9 Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. Association for Computing Machinery. doi:10.1145/1476589.1476628.

10 Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems*, pages 205–231, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46669-8_9.

11 Jeff Smits, Gabriël D. P. Konat, and Eelco Visser. Constructing hybrid incremental compilers for cross-module extensibility with an internal build system. *The Art, Science, and Engineering of Programming*, 4(3), 2020. doi:10.22152/programming-journal.org/2020/4/16.

12 Hendrik van Antwerpen. tree-sitter-stack-graphs-typescript library. URL: https://github.com/github/stack-graphs/tree/main/languages/tree-sitter-stack-graphs-typescript.

13 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

14 Hendrik van Antwerpen and Douglas A. Creager. tree-sitter-stack-graphs library, version 0.6.0, January 2023. doi:10.5281/zenodo.7534898.

15 Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 49–60, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2847538.2847543.

16 Hendrik van Antwerpen, Rob Rix, Douglas A. Creager, et al. tree-sitter-graph 0.7.0, October 2022. doi:10.5281/zenodo.7221982.

17 Aron Zwaan and Hendrik van Antwerpen. Scope graphs: The story so far. In Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann, editors, *Eelco Visser Commemorative Symposium (EVCS 2023)*, pages 13:1–13:13. OpenAccess Series in Informatics, April 2023. doi:10.4230/OASIcs.EVCS.2023.13.

18 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563303.

Type Theory as a Language Workbench

Jan de Muijnck-Hughes
University of Glasgow, UK

Guillaume Allais
University of St Andrews, UK

Edwin Brady
University of St Andrews, UK

Abstract

Language Workbenches offer language designers an expressive environment in which to create their Domain Specific Languages (DSLs). Similarly, research into mechanised meta-theory has shown how dependently typed languages provide expressive environments to formalise and study DSLs and their meta-theoretical properties. But can we claim that dependently typed languages qualify as language workbenches? We argue yes!

We have developed an exemplar DSL called Vélo that showcases not only dependently typed techniques to realise and manipulate Intermediate Representations (IRs), but that dependently typed languages make fine language workbenches. Vélo is a simple verified language with well-typed holes and comes with a complete compiler pipeline: parser, elaborator, REPL, evaluator, and compiler passes. Specifically, we describe our design choices for well-typed IR design that includes support for well-typed holes, how Common Sub-Expression Elimination (CSE) is achieved in a well-typed setting, and how the mechanised type-soundness proof for Vélo is the source of the evaluator.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Software verification; Software and its engineering → Functional languages; Software and its engineering → Formal language definitions; Software and its engineering → Domain specific languages; Theory of computation → Lambda calculus; Software and its engineering → Compilers; Theory of computation → Invariants

Keywords and phrases dependent types, language workbenches, idris2, dsl, edsl, intrinsically scoped, well typed, co-De Bruijn

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.9

Supplementary Material We have made Vélo's source available both as a reproducible artifact and freely available source code:
Software (Source Code): `https://github.com/jfdm/velo-lang`
Software (Artifact): `https://doi.org/10.5281/zenodo.7573031`

Funding Jan de Muijnck-Hughes is funded by EPSRC grants: Border Patrol (EP/N028201/1) and AppControl (EP/V000462/1). Guillaume Allais and Edwin Brady are funded by EPSRC grant: Programming as Conversation: Type-Driven Development in Action (EP/T007265/1).

1 Introduction

Language Workbenches, such as Spoofax [28], offer language designers an expressive environment in which to design, implement, and deploy their Domain Specific Languages (DSLs) [16]. Principally speaking a language workbench [15] is a tool that supports: description of a language's *notation* – how we present a language's concrete syntax to users; implementation of a language's *semantics* – how we realise the language's behaviour; and user interaction through an *editor*. Outside of these core criteria, various language workbenches support language validation, testing, and composition.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 9; pp. 9:1–9:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Concurrently, the mechanised meta-theory research programme [7, 1] has seen a wealth of tools and techniques being developed by the programming languages theory community. In particular, dependently typed languages such as Idris 2 [9], Agda [21], and Coq [26] have been widely used to formalise DSLs, and study their meta-theoretical properties. Dependent types allow types to depend on values – that is, types are first class – and provide an expressive environment in which to reason about, and write, our programs. Efforts using dependently typed languages range from studying specific core calculi [4, 24, 10] to building generic reasoning frameworks [25, 3]. These mechanised software verification projects, however, typically stop short of building the frontend that would let users run these verified language implementations. If our verified language implementations type check, we might as well ship them too! By becoming its own implementation language, Idris 2 has successfully demonstrated that this is not an inescapable fate [9]. But can we now claim that dependently typed languages qualify as language workbenches?

Vélo[1] is a minimal functional language that we have realised in Idris 2 to showcase dependently typed techniques to implement and manipulate Intermediate Representations (IRs). This paper introduces Vélo but, most of all, seeks to show that dependently typed languages make fine language workbenches. We address both the core criteria and some optional extensions highlighted by the language workbench challenge [15] for what constitutes a language workbench. Although not all of the optional criteria are met by dependently typed languages, we are convinced that with some additional engineering (taking advantage of existing work, for example Quickchick [18]) more optional criteria can be satisfied.

Another key tenet in language workbenches, such as Spoofax, is the *ease* with which languages can be created. To that same degree, we have developed a series of reusable modules that captures functionality common to many languages, thereby reducing the *boilerplate* required when creating Embedded Domain Specific Languages (EDSLs) in Idris 2.

Although we have made an effort to make dependently typed programming accessible in our presentation, more introductory material is available for the interested reader [29, 8].

2 Introducing Velo

The design behind Vélo is purposefully unsurprising: it is the Simply-Typed Lambda Calculus (STLC) extended with let-bindings, booleans and their conjunction, and natural numbers and their addition. To promote the idea of interactive editing Vélo also supports well-typed holes. Below we show an example Vélo program, which contains a multiply used hole, and an extract from the REPL session that lists the current set of holes.

```
   let b = false
in let double
         = (fun x : nat => (add x x))
in let x = (double ?hole)
in         (double ?hole)
```

```
Velo> :holes
b : Bool
double : Nat -> Nat
----------
?hole : Nat
```

The featherweight language design of Vélo helps us showcase better how we can use dependently typed languages as language workbenches [17]. Regardless of language complexity, Vélo is nonetheless a complete language with a standard compiler pipeline, and REPL. A DSL captures the language's concrete syntax, and a parser turns DSL instances into raw unchecked terms. Bidirectional type checking keeps type annotations to a minimum in the

[1] A reproducible artifact, and the source code, has been provided as supplementary material.

concrete syntax, and helps to better elaborate raw un-typed terms into a set of well-typed IRs: `Holey` to support well-scoped typed holes; and `Terms` the core representation that captures our language's abstract syntax. We present interesting aspects of our IR design in Section 3. Further, elaboration performs standard desugarings that e.g. turns let-bindings into function application thus reducing the size of our core. From the core representation we also provide well-scoped Common Sub-Expression Elimination (CSE) using co-De Bruijn indexing (Section 4), and we provide a verified evaluator to reduce terms to values (Section 5).

3 Language Design

We begin our discussion by detailing the key design rationale on realising the static semantics of Vélo within Idris 2. We have opted to give Vélo an external concrete syntax (a DSL) in which users can write their programs. With dependently typed languages we can also capture the abstract syntax and its static semantics as an intrinsically scoped and typed EDSL directly within the host language [6]. That is to say that the data structure is designed in such a way that we can only represent well scoped and well typed terms and, correspondingly, that our scope- and type- checking passes are guaranteed to have rejected invalid user inputs. To keep the exposition concise, we focus on a core subset of the language. The interested reader can find the whole definition in the accompanying material.

Types are usually introduced using their context free grammar. We present it here on the left-hand side, it gives users the choice between two base types (Nat, and Bool) and a type former for function types $(\cdot \rightarrow \cdot)$. On the right hand side, we give their internal representation as an inductive type in Idris 2.[2]

$$\begin{array}{rcl} t : \text{TYPE} & ::= & \textsc{Nat} \\ & \mid & \textsc{Bool} \\ & \mid & t \rightarrow t \end{array}$$

```
data Ty = TyNat
        | TyBool
        | TyArr Ty Ty
```

Contexts can be similarly given by a context free grammar: a context is either empty (ϵ), or an existing context (Γ) extended on the right with a new type assignment $(x : t)$ using a comma. In Idris 2, we will adopt a nameless representation and so we represent these contexts by using a `SnocList` of types (i.e. lists that grow on the right). Note that the Idris 2 compiler automatically supports sugar for lists and snoc lists: `[1,2,3]` represents a list counting up from `1` to `3` while `[<1,2,3]` is its snoc list pendant counting down. In particular `[<]` denotes the empty snoc list also known as `Lin`.

$$\begin{array}{rcl} \Gamma : \text{CONTEXT} & ::= & \epsilon \\ & \mid & \Gamma, x : t \end{array}$$

```
data SnocList a = Lin
                | (:<) (SnocList a) a
```

Typing Judgements are given by relations, and encoded in Idris 2 using inductive families, a generalisation of inductive types [14]. Each rule will become a constructor for the family, and so every proof $\Gamma \vdash t : a$ will correspond to a term t of type (`Term` Γ a). On the left hand side we present two judgements: context membership and a typing judgement, and on the right we have the corresponding inductive family declarations.

[2] Throughout this article, the Idris 2 code snippets are automatically rendered using a semantic highlighter. Keywords are typeset in `bold`, types in `blue`, data constructors in `red`, function definitions in `green`, bound variables in `purple`, and comments in *grey*.

$\Gamma \ni x : a$

```
data Elem : (gamma : SnocList ty) -> (a : ty) -> Type
```

$\Gamma \vdash t : a$

```
data Term : (gamma : SnocList Ty) -> (a : Ty) -> Type
```

We leave the definition of `Elem` to the next section, focusing instead on `Term`. The most basic of typing rules are axioms, they have no premise and are mapped to constructors with no argument. We use Idris 2 comments (`--`) to format our constructor's type in such a way that they resemble the corresponding inference rule. Here we show the rule stating that 0 is a natural number and its translation as the `Zero` constructor.

$$\frac{}{\Gamma \vdash \mathsf{Zero} : \textsc{Nat}}$$

```
Zero : ------------------
        Term gamma TyNat
```

Then come typing rules with a single premise which is not a subderivation of the relation itself. They are mapped to constructors with a single argument. Here we show the typing rule for variables: given a proof that we have a variable of type a somewhere in the context, we can build a term of type a in said context.

$$\frac{\Gamma \ni x : a}{\Gamma \vdash x : a}$$

```
Var :  Elem gamma a ->
      --------------
       Term gamma a
```

Next, we have typing rules with a single premise which is a subderivation. They are mapped to constructors with a single argument of the inductive family representing the subderivation. Here we show the typing rule for successor: provided that we are given a natural number in a given context, its successor is also a natural number in the same context.

$$\frac{\Gamma \vdash n : \textsc{Nat}}{\Gamma \vdash (\mathsf{Inc}\, n) : \textsc{Nat}}$$

```
Inc :  Term gamma TyNat ->
      ------------------
       Term gamma TyNat
```

Similarly, rules with two premises are translated to constructors with two arguments, one for each subderivation. Here we present the typing rule for application nodes: provided that the function has a function type, and the argument has a type matching the function's domain, the application has a type corresponding to the function's codomain. Note that the context Γ is the same across the whole rule and so the same `gamma` is used everywhere.

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash t : a}{\Gamma \vdash f \mathbin{\$} t : b}$$

```
App :  Term gamma (TyArr a b) ->
       Term gamma a ->
      ------------------------
       Term gamma b
```

Finally, we have a rule where the premise's context has been extended: a function of type $(a \to b)$ is built by providing a term of type b defined in a context extended with a new variable of type a.

$$\frac{\Gamma, x : a \vdash t : b}{\Gamma \vdash (\lambda(x) \cdot t) : a \to b}$$

```
Func :  Term (gamma :< a) b ->
       ------------------------
        Term gamma (TyArr a b)
```

Using this intrinsically typed representation, we can readily represent entire typing derivations. The following example[3] presents the internal representation `Plus2` of the derivation proving that $(\lambda(x)\cdot(\mathsf{Inc}\,(\mathsf{Inc}\,x)))$ can be assigned the type $(\textsc{Nat} \to \textsc{Nat})$.

$$\dfrac{\dfrac{\vdots}{\epsilon\,, x : \textsc{Nat} \vdash (\mathsf{Inc}\,(\mathsf{Inc}\,x)) : \textsc{Nat}}}{\epsilon \vdash (\lambda(x)\cdot(\mathsf{Inc}\,(\mathsf{Inc}\,x))) : \textsc{Nat} \to \textsc{Nat}}$$

```
Plus2 : Term [<] (TyArr TyNat TyNat)
Plus2 = Func (Inc (Inc (Var Here)))
```

By using `Term` as an IR in our compiler we have made entire classes of invalid programs unrepresentable: it is impossible to form an ill scoped or ill typed term. Indeed, trying to write an ill scoped or an ill typed program leads to a static error as demonstrated by the following **`failing`** blocks.[4] In this first example we try to refer to a variable in an empty context. Idris 2 correctly complains that this is not possible.

```
failing "Mismatch between: ?gamma :< TyNat and [<]."

  Ouch : Term [<] TyNat
  Ouch = Var Here
```

In this second example we try to type the identity function as a function from Nat to Bool. This is statically rejected as nonsensical: `TyNat` and `TyBool` are distinct constructors!

```
failing "Mismatch between: TyBool and TyNat."

  Ouch : Term [<] (TyArr TyNat TyBool)
  Ouch = Func (Var Here)
```

Using such intrinsically typed EDSLs we can statically enforce that our elaborators do check that the raw terms obtained by parsing user input are well scoped and well typed. Writing our compiler passes (model-to-model transformations) and evaluation engine (model-to-host transformation) using these invariant-rich IRs additionally ensures that each step respects the language's static semantics. In fact we will describe in Section 5 how we can use our EDSLs to both verify our static semantics whilst describing our dynamic semantics.

For languages equipped with more advanced type systems, that cannot be as easily enforced statically, we can retain some of these guarantees by using a well scoped core language rather than a well typed one. This is the approach used in Idris 2 and it has already helped eliminate an entire class of bugs arising when attempting to solve a metavariable with a term that was defined in a different context [9].

3.1 Efficient De Bruijn Representation

A common strategy for implementing well-scoped terms is to use typed *De Bruijn* indices [13], which are easily realised as an inductive family [14] indicating where in the type-level context the variable is bound.

Concretely, we index the `Elem` family by a context (once again represented as a `SnocList` of kinds) and the kind of the variable it represents.

$\Gamma \ni x : a$

```
data Elem : (gamma : SnocList ty) -> (a : ty) -> Type
```

[3] `Here` will be defined in Section 3.1 as a constructor for the `Elem` family.
[4] Idris 2 only accepts failing blocks if checking their content yields an error matching the given string.

We then match each context membership inference rule to a constructor. The Here constructor indicates that the variable of interest is the most local one in scope (note the non-linear occurrence of $(x : a)$ on the left hand side, and correspondingly of ty on the right).

$$\frac{}{\Gamma , x : a \ni x : a}$$

```
Here : ----------------------
       Elem (gamma :< ty) ty
```

The There constructor skips past the most local variable to look for the variable of interest deeper in the context.

$$\frac{\Gamma \ni x : a}{\Gamma , y : b \ni x : a}$$

```
There :  Elem gamma ty ->
        ---------------------
         Elem (gamma :< _) ty
```

Whilst a valid definition, this approach unfortunately does not scale to large contexts: every Elem proof is linear in the size of the De Bruijn index that it represents. To improve the runtime efficiency of the representation we instead opt to model De Bruijn indices as natural numbers, which Idris 2 compiles to GMP-style unbounded integers. Further, we need to additionally define an AtIndex family to ensure that all of the natural numbers we use correspond to valid indices. We pointedly reuse the Elem names because these Here and There constructors play exactly the same role.

```
data AtIndex : (ty : kind) -> (ctxt : SnocList kind) ->
               (idx : Nat) -> Type where
  Here : AtIndex ty (ctxt :< ty) 0
  There : AtIndex ty ctxt idx -> AtIndex ty (ctxt :< _) (1 + idx)
```

We then define a variable as the pairing of a natural number and an *erased* (as indicated by the 0 annotation on the binding site for prf) proof that the given natural number is indeed a valid De Bruijn index.

```
data IsVar : (ctxt : SnocList kind) -> (ty : kind) -> Type where
    V : (idx : Nat) -> (0 prf : AtIndex ty ctxt idx) -> IsVar ctxt ty
```

Thanks to Quantitative Type Theory [19, 5] as implemented in Idris 2, the compiler knows that it can safely erase these runtime-irrelevant proofs. we now have the best of both worlds: a well-scoped realisation of De Bruijn indices that is compiled efficiently.

Just like the naïve definition of De Bruijn indexing is not the best suited for a practical implementation, the inductive family Term described in Section 3 is not the most convenient to use. We will now see one of its limitations and how we remedied it in Vélo.

3.2 Compact Constant Folding

Software Foundations' *Programming Language Foundations* opens with a constant-folding transformation exercise [23, Chapter 1]. Starting from a small language of expressions (containing natural numbers, variables, addition, subtraction, and multiplication) we are to deploy the semiring properties to simplify expressions. The definition of the simplifying traversal contains much duplicated code due to the way the source language is structured: all the binary operations are separate constructors, whose subterms need to be structurally simplified before we can decide whether a rule applies. The correction proof has just as much duplication because it needs to follow the structure of the call graph of the function it wants to see reduced. The only saving grace here is that Coq's tactics language lets users write scripts that apply to many similar goals thus avoiding duplication in the source file.

In Vélo, we structure our core language's representation in an algebraic manner so that this duplication is never needed. All builtin operators (from primitive operations on builtin types to function application itself) are represented using a single Call constructor which takes an operation and a type-indexed list of subterms.

```
data Term : (ctxt : SnocList Ty) -> Ty -> Type where
  Var : IsVar ctxt ty -> Term ctxt ty
  Fun : Term (ctxt :< a) b -> Term ctxt (TyArr a b)
  Call : {tys : _} -> (operator : Prim       tys  ty)
                   -> (operands : Terms ctxt tys)
                   -> Term               ctxt      ty
```

Here Terms is the pointwise lifting of Term to lists of types. In practice we use the generic All quantifier, but this is morally equivalent to the specialised version presented below:

```
data Terms : (ctxt : SnocList Ty) -> List Ty -> Type where
  Nil : Terms ctxt Nil
  (::) : Term ctxt ty -> Terms ctxt tys -> Terms ctxt (ty :: tys)
```

The primitive operations can now be enumerated in a single datatype Prim which lists the primitive operation's arguments and the associated return type.

```
data Prim : (args : List Ty) -> (ret : Ty) -> Type where
    Zero : Prim []                    TyNat
    Inc  : Prim [TyNat]               TyNat
    App  : Prim [TyArr dom cod, dom]  cod
```

Using Prim, structural operations can now be implemented by handling recursive calls on the subterms of Call nodes uniformly before dispatching on the operator to see whether additional simplifications can be deployed. Similarly, all of the duplication in the correction proofs is factored out in a single place where the induction hypotheses can be invoked.

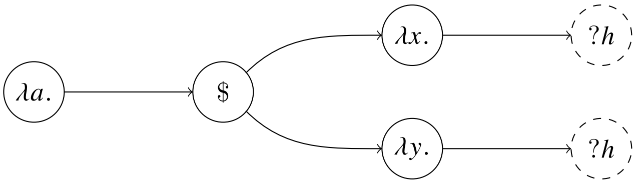
3.3 Well-Typed Holes

Holes are a special kind of placeholder that programmers can use for parts of the program they have not yet written. In a typed language, each hole will be assigned a type based on the context it is used in.

Type-Driven Development [22] is a practice by which the user enters into a dialogue *with* the compiler to interactively build the program. We can enable type-driven programming in part by providing special language support for holes and operations on them. Such operations will include the ability to inspect, refine, compute with, and instantiate (with an adequately typed term) holes. We believe that bare-bones language support for type-driven development should at least include the ability to: (1) inspect the type of a hole and the local context it appears in; (2) instantiate a hole with an adequately typed term; and as well (3) safely evaluate programs that still contain holes. Vélo provides all three.

Idris 2 elaborates holes as it encounters them by turning them into global declarations with no associated definition. Because of this design choice users cannot mention the same hole explicitly in different places to state their intention that these yet unwritten terms ought to be the same. Users can refer to the hole's solution by its name, but that hole is placed in one specific position and it is from that position that Idris 2 infers its context.

In Vélo, however, we allow holes to be mentioned arbitrarily many times in arbitrarily different local contexts. In the following example, the hole `?h` occurs in two distinct contexts: ϵ, a, x and ϵ, a, y.



As a consequence, a term will only fit in that hole if it happens to live in the shared common prefix of these two contexts (ϵ, a). Indeed, references to x will not make sense in ϵ, a, y and vice-versa for y.

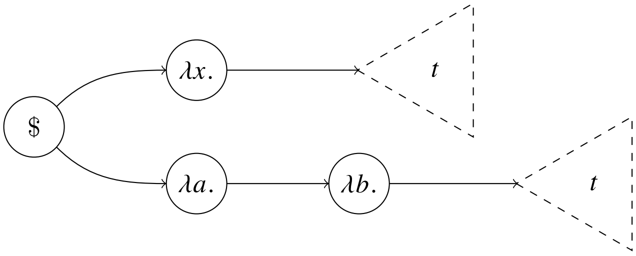
Our elaborator proceeds in two steps. First, a bottom-up pass records holes as they are found and, in nodes with multiple subterms, reconciles conflicting hole occurrences by computing the appropriate local context restrictions. This process produces a list of holes, their types, and local contexts, together with a `Holey` term that contains invariants ensuring these collected holes do fit in the term. Second, a top-down pass produces a core `Term` indexed by the list of `Meta` (a simple record type containing the hole's name, the context it lives in, and its type). Hole occurrences end up being assigned a thinning that embeds the metavariable's actual context into the context it appears in. We discuss thinnings and their use in Vélo in Section 4.

Although these intermediate representations are Vélo-specific, the technique and invariants are general and can be reused by anyone wanting to implement well-scoped holes in their functional DSL.

4 Compiler Passes

Now that our core language is well-scoped by construction, our compiler passes must also be shown to be scope-preserving. This is not a new requirement, merely it makes concrete a constraint that used to be enforced informally. More importantly we show, with our compiler passes, that model-to-same-model transformation of our EDSL is possible, and that the infrastructure required is not bespoke to Vélo.

The purpose of CSE is to identify subterms that appear multiple times in the syntax tree, and to abstract over them to avoid needless recomputations at runtime. In the following example for instance, we would like to let-bind t before the application node (denoted \$) so that t may be shared by both subtrees.



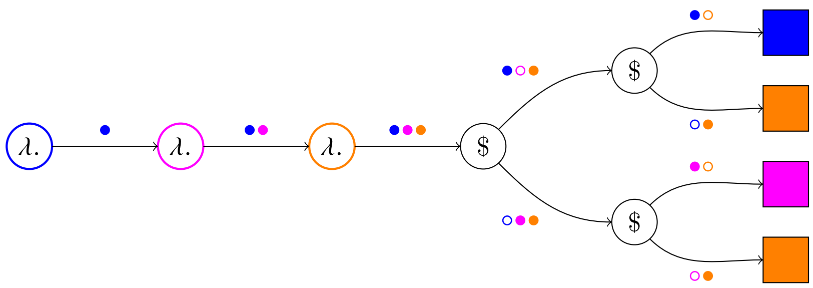
One of the challenges of CSE, as exemplified above, is that the term of interest may be buried deep inside separate contexts. In our intrinsically scoped representation, t in scope Γ, $x : \sigma$ is potentially not actually syntactically equal to a copy living in Γ, $a : \tau$, $b : \nu$. Indeed a variable v bound in Γ will, for instance, be represented by the De Bruijn index $(1 + v)$ in Γ, $x : \sigma$ but by the index $(2 + v)$ in Γ, $a : \tau$, $b : \nu$.

If only terms were indexed by their exact *support* (i.e. a context restricted to the variables actually used in the term)! We would not care about additional yet irrelevant variables that happen to be in scope. The principled solution here is to switch to a different representation when performing CSE. The co-De Bruijn representation [20] provides exactly this guarantee.

In the co-De Bruijn representation, every term is precisely indexed by its exact support. That is to say that every subterm explicitly throws away the bound variables that are not mentioned in it. By the time we reach a variable node, a single bound variable remains in scope: precisely the one being referred to.

This process of throwing unused variables away is reified using thinnings i.e. renamings that are injective, and order preserving. We can think of thinnings as sequences of 0/1 bits, stating whether each variable is kept or dropped.

Below, we give a graphical presentation (taken from [2]) of the S combinator (the lambda term $\lambda g.\lambda f.\lambda x.gx(fx)$) in co-De Bruijn notation. In it we represent thinnings (i.e. lists of bits) as lists of either $\bullet$ (1) or $\circ$ (0).



The first three λ abstractions only use $\bullet$ in their thinnings because all of g, f, and x do appear in the body of the combinator. The first application node then splits the context into two: the first subterm (gx) drops f while the second (fx) gets rid of g. Further application nodes select the one variable still in scope for each leaf subterm: g, x, f, and x.

Using a co-De Bruijn representation, we can identify shared subterms: they need to not be mentioning any of the most local variables and be syntactically equal. Our pass successfully transforms the program on the left-hand side to the one on the right-hand side where the repeated expressions `(add m n)` and `(add n m)` have been let-bound.

```
let m = zero in
let n = (inc zero)
in (add (add (add m n) (add n m))
        (add (add n m) (add m n)))
```

```
let m = zero       in
let n = (inc zero) in
let p = (add n m)  in
let q = (add m n)
in (add (add q p) (add p q))
```

This pass relies on the ability to have a compact representation of thinnings (as the co-De Bruijn representation makes heavy use of them), and additionally the existence of a cheap equality test for them. This is not the case in the implementation we include in Vélo but it is a solved problem [2].

5 Execution

The Vélo REPL lets users reduce terms down to head-normal forms. We can realise Vélo's dynamic semantics either through definitional interpreters [4, 6], or by providing a more traditional syntactic proof of type-soundness [30] but mechanised [29, Part 2: Properties] using inductive families.

We chose the latter approach: by using inductive families, we can make explicit our language's operational semantics. This enables us to study its meta-theoretical properties and in particular prove a progress result: every term is either a value or can take a reduction step. By repeatedly applying the progress result, until we either reach a value or the end user runs out of patience and kills the process, this proof freely gives us an evaluator that is guaranteed to be correct with respect to Vélo's operational semantics.

Following existing approaches [29, Part 2: Properties], we have defined inductive families describing how terms reduce.

```
data Redux : (this,that : Term ctxt type) -> Type where
  SimplifyCall : (op   : Prim tys ty)
              -> (step : Reduxes these those)
                      -> Redux (Call p these) (Call p those)

  ReduceFuncApp : {body  : Term (ctxt :< type) return}
               -> {arg   : Term ctxt type}
               -> (value : Value arg)
                        -> Redux (Call App [Fun body, arg])
                                 (subst arg body)
```

As can be seen above, our setting enforces call-by-value: as described by the rule ReduceFuncApp $((\lambda(x)\cdot b)\,\$\,t)$ only reduces to $(b\,\{x \leftarrow t\})$ if t is already known to be a value. Furthermore, our algebraic design (Section 3.2) allows us to easily enforce a left-to-right evaluation order by having a generic family describing how primitive operations' arguments reduce. As can be seen below: when considering a type-aligned list of arguments, either the hd takes a step and the rest is unchanged, or the hd is already known to be a value and a further argument is therefore allowed to take a step.

```
data Reduxes : (these, those : Terms ctxt tys) -> Type where
  (!:) : (hd   : Redux this that)
      -> (rest : Terms ctxt tys)
              -> Reduxes (this :: rest) (that :: rest)

  (::) : (value : Value hd)
      -> (tl    : Reduxes these those)
               -> Reduxes (hd :: these) (hd :: those)
```

We differ, however, from standard approaches by making our proofs of progress generic such that the boilerplate for computing the reflexive transitive closure when reducing terms is tidied away in a shareable module. Our top-level progress definition is thus parameterised by reduction and value definitions:

```
data Progress : (0 value : Pred a) -> (0 redux : Rel a) -> (tm : a) -> Type
  where Done : {0 tm : a} -> (val : value tm) -> Progress value redux tm

        Step : {this, that : a}
            -> (step : redux this that) -> Progress value redux this
```

and the result of execution, which is similarly parameterised, is as follows (where RTList is the type taking a relation and returning its reflexive-transitive closure):

```
data Result : (0 value : Pred a) -> (0 redux : Rel a) -> (this : a) -> Type
  where R : (that : a) -> (val : value that)
         -> (steps : RTList redux this that) -> Result value redux this
```

The benefit of our approach is that language designers need only provide details of what reductions are, and how to compute a single reduction, the rest comes for free. Moreover, with the result of evaluation we also get the list of reduction steps made that can, optionally, be printed to show a trace of execution.

6 Conclusion

We have shown that dependently typed languages satisfy the core requirements from the *Language Workbench Challenge* [15]. Vélo's *notation* as a DSL is, by design, textual, and the internal core bounded by Idris 2's own notation requirements. More importantly the *semantics* (statics and dynamics) of Vélo are verified as part of the implementation thanks to the dependently typed setting. The weakest supported core criteria, unfortunately, is that for *editor* support. Languages created through Idris 2 do not get an editor, they are free form languages which require their parsers and elaborators be hand written. This can change with future investigation. Idris 2 has support for elaborator reflection [11] which provides a vehicle through which deriving parsers and elaborators can happen.

There are, however, more criteria from the language workbench feature model to consider: semantic & syntactic services for editors; testing & debugging; and composability.

With the rise of the Language Server Protocol (LSP) it would be a good idea to look at how we can derive LSP compatible language servers generically, thus addressing the missing provision of the optional semantic and syntactic services. Idris 2 itself provides an *IDE-Protocol*, and there is support for the LSP in Idris 2.

Our languages also do not come with the ability to test and debug their implementation. Some of the features we have presented are fully formalised (e.g. execution), others are only known to be scope-and-type preserving (e.g. CSE). Therefore the dependently typed setting does not mean we do not need testing anymore. Prior work on generators for inductive families [18] should allow us to bring property-based testing [12] to our core passes.

Finally there is language composability. It would be advantageous to support the reuse of existing languages, and their type-systems when designing new ones. This is a hard problem: One has to not only combine their semantics but also the remainder of the workbench tooling. The *language fragments* approach [27] provides a solution to language composability for intrinsically typed definitional interpreters, but this does not extend to workbench tooling. Extending this approach to our definition of semantics based on inductive families and to creating composable workbench tooling is an open problem.

We strongly believe that with future engineering we can satisfy these missing criteria, and make dependently typed languages a mighty fine language workbench.

References

1 Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019. doi:10.1017/S0956796819000170.

2 Guillaume Allais. Builtin types viewed as inductive families. *CoRR*, abs/2301.02194, 2023. doi:10.48550/arXiv.2301.02194.

3 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.*, 31:e22, 2021. doi:10.1017/S0956796820000076.

4 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. *SIGPLAN Not.*, 52(1):666–679, January 2017. doi:10.1145/3093333.3009866.

5 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.

6 Lennart Augustsson and Magnus Carlsson. An Exercise in Dependent Types: A Well-Typed Interpreter. In *Workshop on Dependent Types in Programming, Gothenburg*, 1999.

7 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi:10.1007/11541868_4.

8 Edwin C. Brady. *Type-Driven Development with Idris.* Manning Publications, 2017.

9 Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.

10 James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for fun and profit. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 255–297. Springer, 2019. doi:10.1007/978-3-030-33636-3_10.

11 David R. Christiansen and Edwin C. Brady. Elaborator reflection: extending idris in idris. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 284–297. ACM, 2016. doi:10.1145/2951913.2951932.

12 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

13 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

14 Peter Dybjer. Inductive families. *Formal Aspects Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

15 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches - conclusions from the language workbench challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. doi:10.1007/978-3-319-02654-1_11.

16 Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

17 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

18 Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, 2018. doi:10.1145/3158133.

19 Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

20 Conor McBride. Everybody's got to be somewhere. In Robert Atkey and Sam Lindley, editors, *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018*, volume 275 of *EPTCS*, pages 53–69, 2018. doi:10.4204/EPTCS.275.6.

21 Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.

22 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL):14:1–14:32, 2019. doi:10.1145/3290327.

23 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2020. Version 5.8, http://softwarefoundations.cis.upenn.edu.

24 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.

25 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

26 The Coq Development Team. The coq proof assistant, January 2022. doi:10.5281/zenodo.5846982.

27 Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563355.

28 Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Softw.*, 31(5):35–43, 2014. doi:10.1109/MS.2014.100.

29 Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*, August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

30 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

On Solving Solved Problems

Sebastian Erdweg
Johannes Gutenberg-Universität Mainz, Germany

Abstract

Some problems are considered solved by the research community. But are they really and does that mean we should stop investigating them? In this essay, I argue that "solved" problems often only appear solved on the surface, while fundamental open research problems lurk below the surface. It requires dedicated researchers to discover those open problems by applying the existing solutions and putting them to the test.

2012 ACM Subject Classification Theory of computation → Parsing; Theory of computation → Program analysis

Keywords and phrases Research Methodology, Parsing, Type Checking

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.10

Acknowledgements The idea for this essay occurred during the preparation for the "Discussion with Experts" session on the Specification and Implementation of Type Systems that Eelco Visser and I moderated at ECOOP 2021. One of the points we had put up for discussion was: Are type systems the new "solved problem that isn't solved"?
I want to thank the reviewers for helping me to get the historical facts straight and André Pacak for feedback on draft of this paper.

1 Introduction

A considerable part of programming-language research is about solving problems. Usually, these problems are motivated through an actual or hypothetical application scenario, where the application of a certain programming methodology or technology is inhibited. Scientific progress then results from removing this inhibitor and solving the corresponding problem, which was unsolved until then. This approach of *solving unsolved problems* is ubiquitous in programming-language research, meaning it is reputable and highly regarded by the scientific community.

This essay does not aim to challenge the value of solving unsolved problems. If anything, the author of this essay is a proponent of this approach to research. However, an excessive focus on unsolved problems can be a disservice to our field, because many "solved" problems are in need of further research.

So what is a *solved problem*? Or better, when does the programming-languages community usually consider a problem to be solved? As I am not aware of any existing definition or systematic investigation of this question, I try to provide a useful definition based on my personal perspective.

Solved Problem: A problem is solved if industry-strength implementations of the solution exist, the solutions have been applied in practice many times, and there are no obvious theoretical challenges remaining.

At first glance, it may seem like solved problems are exactly that: solved. Indeed, solved problems can often be recognized by the lack of interest within the research community. And while a focus on unsolved problems may promise faster scientific progress, all scientific progress is good and there is progress to made on many solved problems.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 10; pp. 10:1–10:6
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Many solved problems have *imperfect solutions.* And these imperfections may not be easy to spot. Rather, it sometimes requires years of experience with a problem and heavy experimentation with the status-quo solution to recognize its shortcomings. And it can be difficult to convince the scientific community that new research results are needed. Not many people carry the necessary drive for perfection and the required perseverance to find a solution as elegant as possible. I was fortunate enough to observe and work with one who inhabited these treats like no other: Eelco Visser.

This essay is going to analyze what made Eelco Visser's research on solved problems so successful. Specifically, we will look at two problems that are solved according to the above definition: parsing and type checking. I will discuss why the community considers these problems to be solved, and why we have seen significant scientific progress in recent years nonetheless. That is, how can a problem be solved and unsolved at the same time? Finally, I will reflect on the research strategy that led to the discovery of imperfections in existing solutions and to the pursuit of elegance in new solutions. What can we learn from this to better embrace research on "solved" problems in our research community?

2 The Case of Parsing

Parsing is a solved problem according to the above definition: There are various industry-strength parsing algorithms and parser generators such as Bison or ANTLR . These and other implementations have been applied in practice many times. And there are no obvious theoretical challenges that remain. Thus, by all means, parsing is a solved problem. Or is it?

While parsing may seem a solved problem on the surface, experts in the field regularly discover and resolve fundamental limitations. For example, consider ambiguous context-free grammars. A context-free grammar is ambiguous if it permits multiple derivations for a single word. Ambiguous grammars occur naturally when declaring the syntax of programming languages, for example, due to operator precedence. The traditional (i.e., old-school) way of resolving such ambiguities is by disambiguation of the grammar. That is, the developer rewrites the grammar and introduces auxiliary non-terminals to ensure only the desired tree can be built. The necessary transformation has been taught to compiler students for decades, and it is an obvious nuisance.

Workarounds like the manual grammar rewriting are symptomatic for "solved" problems, because they highlight the shortcomings of existing solutions. It seems that shortcomings like this are often regarded as *engineering issues* rather than *research problems* by the scientific community. And it takes researchers with a strive for perfection to dedicate themselves to finding elegant solutions to these shortcomings nonetheless. Only in retrospect, it becomes clear that there were research problems to be solved all along.

The problem of ambiguities in context-free grammars was an evident shortcoming of parser generators. In the 1970s, researchers proposed a solution based on disambiguation annotations such as `left` for left-associativity and `right` for right-associativity, and operator priorities to resolve operator precedence [9, 2]. Developers can add disambiguation annotations to their grammar to disambiguate the grammar declaratively. Klint and Visser [14, 21] provided a generalized semantics to such annotations based on parse-tree filtering. Disambiguation annotations were supported in the original SDF framework [12], adopted by Visser in his parsing framework SDF2 [22], and later adopted by other approaches such as ANTLR. And with that, the problem of parsing was considered solved again.

Only much later did parsing researchers discover that the problem of ambiguous grammars was, in fact, not actually solved. Afroozeh et al. found that it was not possible to specify a grammar for OCaml with the available parsing technology [1]. While the previously supported disambiguation only supports shallow priority conflicts, the new research suggested that real-world languages involve *deep priority conflicts* to be solved by disambiguation [1, 7]. The new research also showed that these deep priority conflicts occur often enough in practice to require a general and elegant solution [6]. So yet again, dedicated research teams set out to solve a "solved" problem.

This section so far has mostly retraced the development of declarative disambiguation in context-free parsing. But actually, despite parsing being a solved problem, issues with parsing have been discovered time and again. To name a few, layout-sensitive languages could not be parsed [11, 5], syntactic errors routinely were considered fatal errors that aborted parsing [3], performance of generated parsers often are an order of magnitude slower than hand-written parsers, and incremental parsing remains unattainable for many parser generators.

From this brief discussion, we can conclude that parsing is a "solved" problem that is not actually solved. Before discussing why this discrepancy exists and what we can do about it, let us look at another "solved" problem.

3 The Case of Type Checking

In contrast to type theory and type system design, type checking is a solved problem according to the above definition: There are various industry-strength type checker implementations in compilers and IDEs. These implementations have been applied in practice many times. And there are no obvious theoretical challenges that remain. Thus, by all means, type checking is a solved problem. Or is it?

While type checking may seem a solved problem on the surface, experts in the field regularly discover and resolve fundamental limitations. For example, consider how to implement an *incremental* type checker. A type checker is incremental if it reacts to changes of the checked source code rather than reanalyzing the entire program. Incremental type checkers are crucial in production IDEs and routinely used by compilers that support separate compilation. Nonetheless, a language engineer who wants to implement an incremental type checker from scratch will find little guidance in the literature.

For example, when the Rust compiler and its type checker were incrementalized, the language developers decided to transition the entire compiler from a pass-based to a demand-driven compiler architecture.[1] The demand-driven compiler architecture is based on queries that can depend on each other to form a dependency graph. This compiler and its type checker can now react to source-code changes. For example, when the return type of a function changes, this invalidates queries that depended on the old return type. The invalidated queries are then rerun to produce new type-checking results consistent with the up-to-date source code.

To the Rust developers the lack of incremental type checking support was an evident shortcoming in the field. But the question is this: Are we witnessing *engineering issues* or are there fundamental *research problems* to be addressed by the scientific community? For incremental type checking, a few researchers have been able to repeatedly convince the programming-language community of the need for a systematic approach to incremental type checking. Led by Eelco Visser, Wachsmuth et al. [23] presented an incremental task engine for name and type resolution that was integrated into Spoofax. This approach tracks dependencies between tasks, invalidates task results when an input changes, and reruns tasks much like Rust does nowadays. And almost ten years later, Spoofax features a new

[1] `https://rustc-dev-guide.rust-lang.org/queries/incremental-compilation-in-detail.html`

incremental type checking approach based on scope graphs [24], which the team of Eelco Visser developed in the meantime. This goes to show what kind of continuity and perseverance is necessary to develop new solutions for "solved" problems. Other proposed solutions to incremental type checking include co-contextual typing [10, 15], which rewrites type rules into a form that entails fewer dependencies between type-checking queries, and the compilation of type rules to an incremental Datalog [18], which offloads the incrementalization challenges to incremental Datalog solvers. Only in retrospect, it has become clear that incremental type checking imposes fundamental research problems to be solved.

Beside type checker performance, there are other issues in type checking whose fundamental challenges are gradually becoming more apparent. One of the issues is type-aware editor services, such as code completion. In the state of the art, editor services are developed independent of the language semantics, leading to ad-hoc implementations, development overhead, and potential inconsistencies. Recently, researchers are investigating how to systematically generate type-aware editor services based on the language's syntax and static semantics [4, 17, 19]. And some of the above-mentioned research has led to novel foundations for type checking, such as scope graphs [16] and scope states [20].

In summary, we must acknowledge that type checking is a "solved" problem that is not actually solved. The subsequent section concludes this essay and proposes a way to embrace research on "solved" problems.

4 Why and How to Work on Solved Problems

As the cases of parsing and type checking show, problems regularly appear to be solved when, really, there is plenty of research work to be done. And I think there is a good reason, why researchers tend to mark problems as solved all too early: We can only discover the non-obvious imperfections of existing solutions when applying them to realistic workloads, in realistic contexts, with realistic customers. Of course, this kind of application is hard work; it is not enough to reach for the low-hanging fruits. When exercising research results in realistic applications, we must harvest all the fruits, no matter if they are low-hanging or high up in the tree. Realistic applications help us reveal the high-up research problems that many people do not even notice.

For example, consider Eelco Visser's continuous work on and application of the parsing framework SDF2, which was originally described in his dissertation in 1997 [22]. SDF2 has been applied over and over again to real-world use cases, tested with actual customers, and integrated into the language workbench Spoofax [13] in 2010. And only then did the work on SDF3 start, with the most recent publication stemming from 2020 [8]. Note how the same research team has worked and published on SDF for more than 20 years. In my experience, this is extremely rare: Very few research projects in the programming-language community follow such a long-term trajectory and most research projects are terminated after a few years only.

So it is no surprise that Eelco Visser ended up working on many solved problems. He had to solve solved problems because he put the existing solutions to the test. This way, he was able to observe the shortcomings of those solutions first-hand. And he had that inspiring drive for elegance and discontent for workarounds that made him strive for finding new and better ways. His simultaneous success in academia and his strong industrial collaborations are more than impressive.

So, why and how to work on solved problems? The answer to both questions is the same: Work on realistic applications. We should work on realistic applications to deliver research results that are application-ready, in particular for customers without large R&D

departments of their own. If instead we leave the application to customers, they are driven away from cutting-edge research results when research problems emerge, defaulting to more conservative solutions. Finding and resolving such research problems is our job, we should embrace it.

References

1 Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen J. Vinju. Safe specification of operator precedence rules. In *Software Language Engineering – 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2013.

2 Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. *Commun. ACM*, 18(8):441–452, 1975. doi:10.1145/360933.360969.

3 Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems*, 34(4):15, 2012. doi:10.1145/2400676.2400678.

4 Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 – November 1, 2016*, pages 163–175. ACM, 2016.

5 Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 3–15. ACM, 2018.

6 Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Deep priority conflicts in the wild: a pilot study. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 55–66. ACM, 2017. doi:10.1145/3136014.3136020.

7 Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Towards zero-overhead disambiguation of deep priority conflicts. *Programming Journal*, 2(3):13, 2018. doi:10.22152/programming-journal.org/2018/2/13.

8 Luis Eduardo de Souza Amorim and Eelco Visser. Multi-purpose syntax definition with SDF3. In Frank S. de Boer and Antonio Cerone, editors, *Software Engineering and Formal Methods – 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*, volume 12310 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2020. doi:10.1007/978-3-030-58768-0_1.

9 Jay Earley. Ambiguity and precedence in syntax description. *Acta Informatica*, 4:183–192, 1974. doi:10.1007/BF00288747.

10 Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 880–897. ACM, 2015. doi:10.1145/2814270.2814277.

11 Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2012.

12 Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989. doi:10.1145/71605.71607.

13 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 444–463. ACM, 2010.

14 Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20. Milan, Italy, 1994.

15 Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. A co-contextual type checker for featherweight java. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.ECOOP.2017.18`.

16 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems – 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. `doi:10.1007/978-3-662-46669-8_9`.

17 André Pacak and Sebastian Erdweg. Generating incremental type services. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, pages 197–201. ACM, 2019. `doi:10.1145/3357766.3359534`.

18 André Pacak, Sebastian Erdweg, and Tamás Szabó. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.*, 4(OOPSLA):127:1–127:28, 2020. `doi:10.1145/3428195`.

19 Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. `doi:10.1145/3527329`.

20 Hendrik van Antwerpen and Eelco Visser. Scope states: Guarding safety of name resolution in parallel type checkers. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.1`.

21 Eelco Visser. A case study in optimizing parsing schemata by disambiguation filters. In Sandiway Fong, editor, *International Workshop on Parsing Technologies, IWPT 1997, Boston, MA, USA, September 17-20, 1997*, pages 210–224, 1997. URL: `https://aclanthology.org/1997.iwpt-1.24/`.

22 Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

23 Guido Wachsmuth, Gabriël Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering – 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013. `doi:10.1007/978-3-319-02654-1_15`.

24 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. `doi:10.1145/3563303`.

Reasoning About Paths in the Interface Graph

Michael Greenberg
Stevens Institute of Technology, Hoboken, NJ, USA

Abstract

Clearly specified interfaces between software components are invaluable: development proceeds in parallel; implementation details are abstracted away; invariants are enforced; code is reused. But this abstraction comes with a cost: well chosen interfaces let related tasks be grouped together, but a running program interleaves tasks of all kinds. Reasoning about which values cross a given interface or which interfaces a value will cross is challenging.

It is particularly hard to know that interfaces apply runtime enforcement mechanisms correctly: as programs run, values cross abstraction boundaries in subtle ways. One particular case of such reasoning – proving that a contract system checks contracts correctly at runtime [2, 3] – uses a dynamic analysis to keep track of which interfaces are responsible for which values. The dynamic analysis works by giving an alternative semantics that "colors" values to match the components responsible for them. No program is ever *run* in this alternative semantics – it's a formal tool to verify that the contract system's enforcement is correct.

In this short paper, we refine Dimoulas et al.'s dynamic analysis to more precisely track colors, phrasing our results graph theoretically: a value's colors are a path in the interface graph of the original program. Our graph theoretic framing makes it easy to see that the dynamic analysis is subsumed by Eelco Visser's scope graphs.

2012 ACM Subject Classification Software and its engineering → Functional languages; Mathematics of computing → Graph theory; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases interfaces, components, lambda calculus, dynamic analysis

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.11

Acknowledgements I gratefully acknowledge the reviewers for their advice.

1 Introduction

Clearly specified interfaces between parts of a program – modules, libraries, components, etc. – are key to software development. Good interfaces lighten the load of design, development, and maintenance; good abstractions between interfaces make it easier to reason about programs. Clever uses of module boundaries can yield dynamic guarantees: contracts on interfaces help programmers identify modules that violate preconditions or don't live up to postconditions; object capability interfaces help programmers moderate access to critical resources.

But nicely abstracted, compartmentalized code doesn't *stay* nicely abstracted and compartmentalized at runtime: at runtime, our beautiful abstraction barriers collapse into a dynamic morass. Proving that a static enforcement mechanism yields worthwhile guarantees can be hard – but it isn't even clear where to start for dynamic enforcement mechanisms. What property must we even prove to know that our runtime checks mean nothing goes wrong?

Dimoulas et al. [2] use a dynamic analysis to prove that their higher-order contracts check values appropriately. They describe a simple functional language – Contract PCF (CPCF) – that puts *contract monitors* at interfaces, e.g., a math module might export a square-root function with the contract $\{x : \mathsf{Float} \mid x \geq 0\} \to \{y : \mathsf{Float} \mid \mathsf{abs}(y^2 - x) < \epsilon\}$, requiring that its clients give non-negative inputs and promising to return square roots (within some constant ϵ). At first order, these contracts amount to pre- and post-conditions. At higher orders, *blame labels* identify the component responsible for contract failures, following Findler

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 11; pp. 11:1–11:11
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and Felleisen [4]. For Dimoulas et al., the core question is: if a value fails a check, does the contract blame the correct component? Their dynamic analysis tracks which components are "responsible" for each value. Each component has a "color" (a unique label, used also for blame); values are colored with the components that are responsible for them. Using bisimulation techniques, they prove that when a value fails a contract check, the label blamed, the value that failed is colored with the same label – that is, when a value fails a check, we blame the responsible component.

Dimoulas et al.'s approach is clever: the dynamic analysis annotates values with crucial information... but rather than actually *running* the dynamic analysis, they prove a theorem about it. Their approach is also complex: their analysis is specific to contracts; their analysis is doesn't precisely track the order of components responsible for a value. **In this short paper, we extend and generalize Dimoulas et al.'s analysis to track how values dynamically traverse a program's static *interface graph*.** Our extension (Section 2) yields a more precise dynamic analysis for reasoning about a program's components graph theoretically (Section 3). Our more precise analysis yields tools for relating values and program control flow back to a program's interface graph (Section 4). Finally, our notion of interface graph is not dissimilar to Eelco Visser's scope graphs [8] (Section 5).

2 A dynamic analysis for components and responsibility

Behavioral contracts can specify as little as the type of a function or as much as the full behavior of a component through time. In first-order languages, contracts amount to pre- and post-conditions; in higher-order languages, contracts must carefully track *blame* in order to account for the flow of higher-order values. Findler and Felleisen [4] show that it suffices to have just two blame labels to track a higher-order value: a "positive" label for the value itself, and "negative" label for the context.

Dimoulas et al. study higher-order behavioral contracts, with the aim of proving that contract violations blame the correct party. They extend the middleweight PCF calculus [9] to Contract PCF (CPCF), adding a notion of contracts to monitor the interfaces between components. Their key insight is that it's possible to run the program in a semantics that uses a dynamic analysis to track the parties responsible for each value, devising a semantics that: "(1) for each party, keeps track of its contract obligations and (2) for each value, accounts for its origin" [2]§3. That is, a contract system is correct when every time it checks a value for contract conformance at some interface, then that interface is now "responsible for" that value. To fulfill (1), each predicate contract – like `number?` or `nonempty-list?` – keeps track of a set of responsibilities, i.e., the *colors* (unique labels) of those values it has monitored. To fulfill (2), expressions and values keep track of a vector of colors identifying the components responsible for that expression. Conflating blame labels and component colors, Dimoulas et al. use this notion of correctness to great effect: they discover the need for a new *indy* semantics that carries a *third* label in case contracts themselves violate preconditions (writing monitors as $\mathsf{mon}^{l_1,l_2}_{l_3}$, where l_3 refers to the contract); they show that the *picky* semantics for contracts cannot be correct in Racket [2], while the *lax* one can miss violations [3].

We refine Dimoulas et al.'s Contract PCF into the *colored lambda calculus*, the Clam calculus for short. In doing so, we separate their tool – the dynamic analysis in the semantics – from their use of it – proving correctness for contracts. Along the way, we refine their analysis to make it more precise.

$$\begin{array}{llll}\textbf{Terms} & e & ::= & x \mid k \mid op(e_1, \dots, e_n) \mid \lambda x.\ e \mid e_1\ e_2 \mid \textbf{if}\ e_1\ e_2\ e_3 \mid \\ & & & \mathsf{ifc}(e)^{l_1,l_2} \mid \|e\|^{l_1,l_2} \\ \textbf{Labels} & l & \in & \mathsf{Label} = \{l_0, l_1, \dots\}\end{array}$$

Figure 1 Syntax for the Clam calculus.

2.1 Syntax

The Clam calculus extends untyped lambda calculus with conventional notions of constant, first-order operation, and conditional control flow – along with two new forms for marking interfaces between components and colored expressions (Figure 1). These two extensions, borrowed from Dimoulas et al., bear further discussion. Interfaces connect *components* (deliberately generically named); we conflate components and their colors/labels, drawn from some infinite set, Label. An interface form $\mathsf{ifc}(e)^{l_1,l_2}$ represents a static interface boundary: l_2 is the label for the "outside" of the interface, and l_1 is the label for the inside of the interface, which is implemented by the term e. These interfaces are *static*, occurring in the original program text. A colored expression $\|e\|^{l_1,l_2}$ represents dynamically determined responsibility: the component colored l_1 was previously responsible for the expression e, but now the component colored l_2 is responsible. These colored expressions are *dynamic*, arising as values flow through interfaces. Our syntax is directly adapted from Dimoulas et al.'s original framing [2]: we've removed the contract checks, renaming their monitors mon into our interfaces ifc. By removing contract checks, we no longer need the third blame label; we track a *list* of labels on interfaces rather than their set of obligations on a predicate contract, allowing us to characterize values using more precise, graph-theoretic notions (Section 3). Finally, a colored expression gives both an inner and an outer label, unlike Dimoulas's single label. To put the last difference another way, our notion of responsibility is always expressed as a *transfer*; when just the component colored l is responsible for the un-transferred expression, we write $\|e\|^{l,l}$.

We use standard encodings of notations like let; given a finite series of colors l_n, we write the left-to-right multi-coloring of an expression as: $\|e\|^{\overrightarrow{l_n}} = \|\|\|\|e\|^{l_1,l_2}\|^{l_2,\dots}\|^{\dots,l_{n-1}}\|^{l_{n-1},l_n}$ where l_n is the outermost color. When the list of colors is empty, $\|e\|^{\epsilon} = e$.

A *source program* has interfaces but no colored expressions: that is, a source program's components are connected, but no component has yet taken responsibility for the value.

We conflate a component and its label, but a component's constituent expressions may be scattered through a term. For example, consider the following term e_0:

$$\begin{array}{lcl} e_0 & = & \mathsf{ifc}(\mathsf{ifc}(e_1)^{l_2,l_1}\ 1)^{l_1,l_0} \\ e_1 & = & \mathsf{ifc}(\mathsf{ifc}(e_2)^{l_1,l_3})^{l_3,l_2} \\ e_2 & = & \lambda x.\ x + 1 \end{array}$$

By convention, l_0 is the outermost label; the component l_1 applies $\mathsf{ifc}(\dots)^{l_2,l_1}$ to the value 1 – and deeply nested inside, the same component l_1 contains the function $\lambda x.\ x + 1$. As another example, consider the following term, e_3:

$$\begin{array}{lcl} e_3 & = & \mathsf{ifc}(\mathsf{ifc}(e_4)^{l_m,l_c}\ \dots)^{l_c,l_0} \\ e_4 & = & \lambda x.\ \mathsf{let}\ db = \mathsf{ifc}(e_5)^{l_d,l_m}\ \mathsf{in} \\ & & \quad \textbf{if}\ (\mathit{fst}\ db = x)\ (\mathit{snd}\ db)\ (-1) \\ e_5 & = & (\mathsf{password}, \mathsf{secret}) \end{array}$$

Here, the component l_c applies the results of component l_m to some unspecified value ($\dots$ in e_3); the component l_d holds the database (e_5) and is used by component l_m (e_4).

$$
\begin{array}{lrcl}
\textbf{Values} & v & ::= & \|u\|^{l_1,l_2} \mid \|v\|^{l_1,l_2} \\
\textbf{Pre-values} & u & ::= & k \mid \lambda x.\ e \\
\textbf{Frames} & Fr & ::= & \bullet\, e \mid v\, \bullet \mid op(v_1,\,\dots,\,v_i,\,\bullet,\,e_1,\,\dots,\,e_j) \mid \textbf{if}\ \bullet\ e_2\ e_3 \mid \\
 & & & \mathsf{ifc}(\bullet)^{l_1,l_2} \mid \|\bullet\|^{l_1,l_2} \\
\textbf{Continuations} & C & ::= & \bullet \mid C{::}Fr \\
 & & & \\
\textbf{Machine states} & st & ::= & \langle C, l, e\rangle
\end{array}
$$

Figure 2 Syntax definitions for the Clam abstract machine.

$$
\begin{array}{lll}
\langle C, l, e_1\ \ e_2\rangle & \longrightarrow \langle C{::}\bullet\ \ e_2, l, e_1\rangle & \textsc{AppStart} \\
\langle C{::}\bullet\ \ e_2, l, \|\lambda x.\ e_1\|^{l_1,\dots,l_n}\rangle & \longrightarrow \langle C{::}\|\lambda x.\ e_1\|^{l_1,\dots,l_n}\bullet, l, e_2\rangle & \textsc{AppMiddle} \\
\langle C{::}\|\lambda x.\ e_1\|^{l_1,\dots,l_n}\bullet, l, \|u_2\|^{l'_1,\dots,l'_m}\rangle & \longrightarrow \langle C, l, \|e_1\{\|u_2\|^{l'_1,\dots,l'_m,l_n,\dots,l_1}/x\}\|^{l_1,\dots,l_n}\rangle & \textsc{AppEnd} \\
\langle C, l, op(e_1,\, e_2,\, \dots,\, e_n)\rangle & \longrightarrow \langle C{::}op(\,\bullet\,,\, e_2,\, \dots,\, e_n), l, e_1\rangle & \textsc{OpStart} \\
\langle C{::}op(v_1,\, \dots,\, v_i,\, \bullet\,,\, e_1,\, e_2,\, \dots,\, e_j), l, v\rangle & \longrightarrow \langle C{::}op(v_1,\, \dots,\, v_i,\, v,\, \bullet\,,\, e_2,\, \dots,\, e_j), l, e_1\rangle & \textsc{OpMiddle} \\
\langle C{::}op(\|u_1\|^{\overrightarrow{l_1}},\, \dots,\, \|u_i\|^{\overrightarrow{l_i}},\, \bullet\,), l, \|u\|^{\overrightarrow{l}}\rangle & \longrightarrow \langle C, l, [\![op]\!]\,(u_1,\, \dots,\, u_i,\, u)\rangle & \textsc{OpEnd} \\
\langle C, l, \textbf{if}\ \ e_1\ \ e_2\ \ e_3\rangle & \longrightarrow \langle C{::}\textbf{if}\ \bullet\ \ e_2\ e_3, l, e_1\rangle & \textsc{IfStart} \\
\langle C{::}\textbf{if}\ \bullet\ \ e_2\ e_3, l, \|\mathsf{true}\|^{l_1,\dots,l_n}\rangle & \longrightarrow \langle C, l, e_2\rangle & \textsc{IfEndTrue} \\
\langle C{::}\textbf{if}\ \bullet\ \ e_2\ e_3, l, \|\mathsf{false}\|^{l_1,\dots,l_n}\rangle & \longrightarrow \langle C, l, e_3\rangle & \textsc{IfEndFalse} \\
\langle C, l, u\rangle & \longrightarrow \langle C, l, \|u\|^{l,l}\rangle & \textsc{ColorValue} \\
\langle C, l, \|e\|^{l_1,l_2}\rangle & \longrightarrow \langle C{::}\|\bullet\|^{l_1,l_2}, l, e\rangle\ \text{when}\ e \neq \|u\|^{\dots,l'} & \textsc{ColorStart} \\
\langle C{::}\|\bullet\|^{l_1,l_2}, l, v\rangle & \longrightarrow \langle C, l, \|v\|^{l_1,l_2}\rangle & \textsc{ColorEnd} \\
\langle C, l, \mathsf{ifc}(e)^{l_1,l_2}\rangle & \longrightarrow \langle C{::}\mathsf{ifc}(\bullet)^{l_1,l}, l_1, e\rangle & \textsc{IfcStart} \\
\langle C{::}\mathsf{ifc}(\bullet)^{l_1,l_2}, l, \|k\|^{l'_1,\dots,l'_n}\rangle & \longrightarrow \langle C, l_2, \|k\|^{l'_1,\dots,l'_n,l_2}\rangle & \textsc{IfcEndBase} \\
\langle C{::}\mathsf{ifc}(\bullet)^{l_1,l_2}, l, \|\lambda x.\ e\|^{l'_1,\dots,l'_n}\rangle & \longrightarrow \langle C, l_2, \lambda x.\ \mathsf{ifc}(\|\lambda x.\ e\|^{l'_1,\dots,l'_n}\ \mathsf{ifc}(x)^{l_2,l_1})^{l_1,l_2}\rangle & \textsc{IfcEndLam}
\end{array}
$$

Figure 3 Reduction semantics for the Clam abstract machine.

2.2 Semantics

While Dimoulas et al. use small-step operational semantics to give meaning to CPCF, we use an abstract machine (Figures 2 and 3). Why? It's hard to track the "current" component in a small-step operational semantics, while an abstract machine makes it easy – essential for our graph theoretic properties (Sections 3 and 4).

We define the Clam abstract machine in terms of *machine states st*, which are a triple of the current *continuation* C (made up of *frames* Fr), the current *component label* l, and the current term to reduce e (Figure 2). Our semantics distinguishes between colorless pre-values u and colored values v. In the Clam calculus, it is an invariant that a colored expression's last color is the current component: for the component to be computing with a value, it must be responsible for it. Reduction rules map machine states to machine states (Figure 3). The most interesting rules are the ones for coloring and interfaces. The current component changes whenever control enters a (static) interface or (dynamic) colored expression (IfcStart, ColorStart); the component reverts back when control leaves (IfcEnd*, ColorEnd). The ColorValue rule colors pre-values with the current component, dynamically making the current component responsible for any values it generates.

Like in Dimoulas et al.'s semantics, interfaces treat constants and functions differently when they pass over interfaces. When a constant moves to a new component, it acquires the color of that component (IfcEndBase); when a function moves to a new component, it is wrapped in a *function proxy* that takes the interface with it (IfcEndLam). That is, a function f moving from l_1 to l_2 is wrapped in a lambda so that arguments to f in the component l_2 are sent back to l_1 before running the function, whose result is moved from l_1 to l_2. Put another way, interfaces don't add colors to lambdas at all – instead, interfaces generate wrappers that do the "real" work on the arguments. The idea here – borrowed directly from Dimoulas et al. – is that when higher-order values move from one component to another, their bodies remain in the old component and values passed in to them come from the new one.

One might wonder: why have two notions of coloring? First, it is convenient for reasoning to distinguish static interfaces $\mathsf{ifc}(\bullet)^{l_1,l_2}$ from dynamic ones $\| \bullet \|^{l_1,l_2}$. More importantly, it is technically important to treat higher-order values specially at static interfaces (IfcEndLam).

Like in Dimoulas et al.'s semantics, our β reduction rule (AppEnd) updates responsibility on substitution. Unlike their semantics, we don't add the current component to the substituted value: the current container l is the same as the final color l_n on the function and the final color l'_m on the argument, so there's no need to add colors from the context. This redundancy doesn't quite exist in their system. Since our coloring brackets have ordered pairs of colors, we have a more precise account of inner and outer colors. Furthermore, we also color the substituted argument value with the color of the lambda – as it has substituted in for the variable in the lambda – and whoever is responsible for the lambda is responsible for its input parameter. Both systems have redundancy in them: values will collect many redundant copies of the same color. These redundant colors correspond to self-loops in the interface graph between components (Section 3). Rather than treating a values accumulated colors as an arbitrary list, we could have colors form a monoid with an operation that deduplicates self loops. There's no formal benefit in our abstract setting, though, so we don't bother.

Also following Dimoulas et al., operations do not taint their outputs (OpEnd): operations strip the colors off their inputs and produce uncolored prevalues.

In general, we'll assume that there's some default outer label l_0, and to run the program e we start with the configuration $\langle \bullet, l_0, e \rangle$. We'll only do so, however, if e is well colored at l_0.

2.3 Well colored terms

An expression is well colored when its interfaces and colored subexpressions agree (Figure 4); rules for frames and continuations follow naturally, with $\vdash C : l \twoheadrightarrow l'$ meaning that the continuation C expects its hole to be filled with something labeled l and produces a term labeled l'.

Coloring enjoys the expected type-like properties of substitution and preservation. Coloring has no progress property: colors don't say anything about the correctness of the program's operations; colors say that a program's components agree at their interfaces. Ill colored programs do *not* go wrong.

▶ **Lemma 2.1** (Substitution). *If $L, x{:}l', L' \vdash e : l$ and $\emptyset \vdash v : l'$ (where $v = \|u\|^{\dots,l'}$), then $L, L' \vdash e\{v/x\} : l$.*

Proof. By induction on the coloring derivation of e, leaving L' general and relying on a weakening lemma. ◀

▶ **Lemma 2.2** (Preservation). *If $\vdash st : l'$ and $st \longrightarrow st'$ then $\vdash st : l'$.*

$$\boxed{L \vdash e : l}$$

$$\frac{x{:}l' \in L}{L \vdash x : l'}\ \textsc{Var} \qquad \frac{}{L \vdash k : l}\ \textsc{Const} \qquad \frac{L, x{:}l \vdash e_{12} : l}{L \vdash \lambda x.\ e_{12} : l}\ \textsc{Abs} \qquad \frac{L \vdash e_i : l}{L \vdash op(e_1, \ldots, e_n) : l}\ \textsc{Op}$$

$$\frac{L \vdash e_1 : l \quad L \vdash e_2 : l}{L \vdash e_1\ e_2 : l}\ \textsc{App} \qquad \frac{L \vdash e_1 : l \quad L \vdash e_2 : l \quad L \vdash e_3 : l}{L \vdash \textbf{if}\ e_1\ e_2\ e_3 : l}\ \textsc{If}$$

$$\frac{L \vdash e : l_1}{L \vdash \|e\|^{l_1,l_2} : l_2}\ \textsc{Color} \qquad \frac{L \vdash e : l_1}{L \vdash \mathsf{ifc}(e)^{l_1,l_2} : l_2}\ \textsc{Ifc}$$

$$\boxed{\vdash Fr : l_1 \twoheadrightarrow l_2} \quad \boxed{\vdash C : l_1 \twoheadrightarrow l_2} \quad \boxed{\vdash st : l'}$$

$$\frac{\emptyset \vdash e : l}{\vdash \bullet\ e : l \twoheadrightarrow l}\ \textsc{FAppL} \qquad \frac{\emptyset \vdash v : l}{\vdash v\ \bullet : l \twoheadrightarrow l}\ \textsc{FAppR} \qquad \frac{\emptyset \vdash e_2 : l \quad \emptyset \vdash e_3 : l}{\vdash \textbf{if}\ \bullet\ e_2\ e_3 : l \twoheadrightarrow l}\ \textsc{FIf}$$

$$\frac{}{\vdash \|\bullet\|^{l_1,l_2} : l_1 \twoheadrightarrow l_2}\ \textsc{FColor} \qquad \frac{}{\vdash \mathsf{ifc}(\bullet)^{l_1,l_2} : l_1 \twoheadrightarrow l_2}\ \textsc{FIfc} \qquad \frac{}{\vdash \bullet : l \twoheadrightarrow l}\ \textsc{Hole}$$

$$\frac{\vdash C : l' \twoheadrightarrow l'' \quad \vdash Fr : l \twoheadrightarrow l'}{\vdash C{::}Fr : l \twoheadrightarrow l''}\ \textsc{Frame} \qquad \frac{\vdash C : l \twoheadrightarrow l' \quad \emptyset \vdash e : l}{\vdash \langle C, l, e\rangle : l'}\ \textsc{State}$$

Figure 4 Well colored expressions have correct labels on interfaces and colored expressions. Well colored continuations transition according to well colored frames; well colored states match the term to the current component and the continuation.

Proof. By induction on the coloring derivation. Let $st = \langle C, l, e\rangle$. We go first by cases on the coloring of e as $\emptyset \vdash e : l$, considering the continuation C and its coloring $\vdash C : l \twoheadrightarrow l'$ as necessary. ◀

3 Interface and value graphs

The Clam calculus explicitly marks interfaces between components in the program so that we can reason clearly about how components communicate. We use graphs to formalize the idea: a term e has a *program graph* marking the interfaces between each colored component in e. Program graphs are undirected graphs where each color is a node; when there exists an interface between two components/colors, an edge connects their corresponding nodes.

Each term e has two kinds of program graphs (Figure 5): the interface graph and the value graph. In both graphs, the nodes are simply the set of colors used in e, i.e., e's components. In the *interface graph*, $\mathbf{ig}(e)$, we take the graph's edges from interfaces – there is an edge $\{l_1 \leftrightarrow l_2\}$ for each interface $\mathsf{ifc}(e)^{l_1,l_2}$. In the *value graph*, $\mathbf{vg}(e)$, we take the graph's edges from colorings – there is an edge $\{l_1 \leftrightarrow l_2\}$ for each coloring $\|e\|^{l_1,l_2}$. For both graphs, we assume that all nodes have self loops: for every $l \in e$, the edge $\{l \leftrightarrow l\}$ is in every program graph. Both kinds of program graphs are mostly defined homomorphically – the interesting cases come in the treatment of interfaces and colored expressions.

While an expression has a straightforward tree structure, well colored expressions will have interface *graphs*, with loops. The program e_0 from Section 2.1 has a lasso structure (Figure 6, left): there is a loop using the edges $\{l_2 \leftrightarrow l_1\}$ and $\{l_3 \leftrightarrow l_2\}$ and $\{l_1 \leftrightarrow l_3\}$. Evaluating this program yields a value whose colors trace appropriately through the interface graph (Figure 7).

$$
\begin{array}{rcl}
\mathbf{ig}(x) & = & \emptyset \\
\mathbf{ig}(k) & = & \emptyset \\
\mathbf{ig}(\lambda x.\ e) & = & \mathbf{ig}(e) \\
\mathbf{ig}(op(e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} \mathbf{ig}(e_i) \\
\mathbf{ig}(e_1\ e_2) & = & \mathbf{ig}(e_1) \cup \mathbf{ig}(e_2) \\
\mathbf{ig}(\mathbf{if}\ e_1\ e_2\ e_3) & = & \bigcup_{i=1}^{3} \mathbf{ig}(e_i) \\
\mathbf{ig}(\mathsf{ifc}(e)^{l_1,l_2}) & = & \{l_1 \leftrightarrow l_2\} \cup \mathbf{ig}(e) \\
\mathbf{ig}(\|e\|^{l_1,l_2}) & = & \mathbf{ig}(e)
\end{array}
\qquad
\begin{array}{rcl}
\mathbf{vg}(x) & = & \emptyset \\
\mathbf{vg}(k) & = & \emptyset \\
\mathbf{vg}(\lambda x.\ e) & = & \mathbf{vg}(e) \\
\mathbf{vg}(op(e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} \mathbf{vg}(e_i) \\
\mathbf{vg}(e_1\ e_2) & = & \mathbf{vg}(e_1) \cup \mathbf{vg}(e_2) \\
\mathbf{vg}(\mathbf{if}\ e_1\ e_2\ e_3) & = & \bigcup_{i=1}^{3} \mathbf{vg}(e_i) \\
\mathbf{vg}(\mathsf{ifc}(e)^{l_1,l_2}) & = & \mathbf{vg}(e) \\
\mathbf{vg}(\|e\|^{l_1,l_2}) & = & \{l_1 \leftrightarrow l_2\} \cup \mathbf{vg}(e)
\end{array}
$$

Figure 5 Program graphs record the connections between interfaces in a program. The interface graph **ig** records interfaces, while the value graph **vg** records coloring/responsibility relationships. Interesting rows are highlighted in lavender. Definitions for frames can be derived from definitions for expressions, where holes • have empty graphs.

$$
\begin{array}{rcl}
e_0 & = & \mathsf{ifc}(\mathsf{ifc}(e_1)^{l_2,l_1}\ 1)^{l_1,l_0} \\
e_1 & = & \mathsf{ifc}(\mathsf{ifc}(e_2)^{l_1,l_3})^{l_3,l_2} \\
e_2 & = & \lambda x.\ x + 1
\end{array}
\qquad
\begin{array}{rcl}
e_3 & = & \mathsf{ifc}(\mathsf{ifc}(e_4)^{l_m,l_c}\ \ldots)^{l_c,l_0} \\
e_4 & = & \lambda x.\ \mathsf{let}\ db = \mathsf{ifc}(e_5)^{l_d,l_m}\ \mathsf{in} \\
 & & \quad \mathbf{if}\ (\mathit{fst}\ db = x)\ (\mathit{snd}\ db)\ (-1) \\
e_5 & = & (\mathsf{password}, \mathsf{secret})
\end{array}
$$

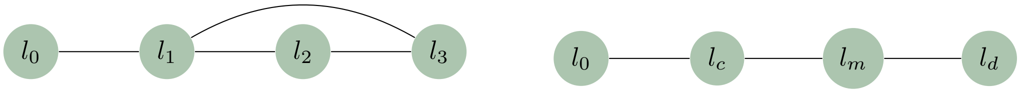


Figure 6 Example programs and their corresponding interface graphs.

But some programs are deliberately structured to avoid certain forms of communication. Say, a client should only ever talk to the database by way of the middleware, never directly (Figure 6, right). The term e_5 is a password-protected database in component l_d, containing a value secret that appears nowhere else in the program. The term e_3 is a client of the database in component l_c – it supplies a password and tries to work with the secret. The term e_4 is the middleware in component l_m; it takes a password from the client and either returns the secret database contents or signals an error by returning -1. By inspecting the interface graph, we can see that there is no direct connection from the database to the client.

4 Paths in the interface graph

Interface and value graphs let us characterize how values flow between components. Inspired by Dimoulas et al.'s proof of contract correctness, we show that (1) *as programs run, their trace is a path in the interface graph* (Lemma 4.1) and (2) *value colorings are paths in the interface graph* (Lemma 4.2). Both properties rely on a subject reduction like lemma: reducing a term yields a new term whose interface graph is a subgraph of the original term's interface graph (Lemma A.3). A similar property holds for the value graph, though the value graph of a term may gain edges as interfaces turn into colored expressions (Figure 3, IfcEnd*).

$$
\begin{array}{rlll}
 & \langle \bullet, & l_0, & \mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2})^{l_2,l_1}\ 1)^{l_1,l_0}\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}, & l_1, & \mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2})^{l_2,l_1}\ 1\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0} :: \bullet\ 1, & l_1, & \mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2})^{l_2,l_1}\rangle \\
\longrightarrow & \langle \ldots :: \bullet\ 1 :: \mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2}\rangle \\
\longrightarrow & \langle \ldots :: \mathsf{ifc}(\bullet)^{l_2,l_1} :: \mathsf{ifc}(\bullet)^{l_3,l_2}, & l_3, & \mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3}\rangle \\
\longrightarrow & \langle \ldots :: \mathsf{ifc}(\bullet)^{l_3,l_2} :: \mathsf{ifc}(\bullet)^{l_1,l_3}, & l_1, & \lambda x.\ x+1\rangle \\
\longrightarrow & \langle \ldots :: \mathsf{ifc}(\bullet)^{l_3,l_2} :: \mathsf{ifc}(\bullet)^{l_1,l_3}, & l_1, & \|\lambda x.\ x+1\|^{l_1,l_1}\rangle \\
\longrightarrow & \langle \ldots :: \mathsf{ifc}(\bullet)^{l_2,l_1} :: \mathsf{ifc}(\bullet)^{l_3,l_2}, & l_3, & \lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\rangle \\
\longrightarrow & \langle \ldots :: \mathsf{ifc}(\bullet)^{l_2,l_1} :: \mathsf{ifc}(\bullet)^{l_3,l_2}, & l_3, & \|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\rangle \\
\longrightarrow & \langle \ldots :: \bullet\ 1 :: \mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \\
\multicolumn{4}{r}{\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\rangle} \\
\longrightarrow & \langle \ldots :: \bullet\ 1 :: \mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \\
\multicolumn{4}{r}{\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0} :: \bullet\ 1, & l_1, & \\
\multicolumn{4}{r}{\lambda w.\ \mathsf{ifc}(\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(w)^{l_1,l_2})^{l_2,l_1}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0} :: \bullet\ 1, & l_1, & \\
\multicolumn{4}{r}{\|\lambda w.\ \mathsf{ifc}(\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \ldots\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(w)^{l_1,l_2})^{l_2,l_1}\|^{l_1,l_1}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0} :: \|\lambda w.\ \ldots\|^{l_1,l_1}\ \bullet, & l_1, & 1\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0} :: \|\lambda w.\ \ldots\|^{l_1,l_1}\ \bullet, & l_1, & \|1\|^{l_1,l_1}\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}, & l_1, & \\
\multicolumn{4}{r}{\mathsf{ifc}(\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \ldots\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(\|1\|^{l_1,l_1})^{l_1,l_2})^{l_2,l_1}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0} :: \mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \\
\multicolumn{4}{r}{\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \ldots\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(\|1\|^{l_1,l_1})^{l_1,l_2}\rangle} \\
\longrightarrow & \langle\langle \ldots :: \mathsf{ifc}(\bullet)^{l_2,l_1} :: \|\lambda z.\ \ldots\|^{l_2,l_2}\ \bullet, & l_2, & \mathsf{ifc}(\|1\|^{l_1,l_1})^{l_1,l_2}\rangle\rangle \\
\longrightarrow & \langle\langle \ldots :: \|\lambda z.\ \ldots\|^{l_2,l_2}\ \bullet :: \mathsf{ifc}(\bullet)^{l_1,l_2}, & l_1, & \|1\|^{l_1,l_1}\rangle\rangle \\
\longrightarrow & \langle\langle \ldots :: \mathsf{ifc}(\bullet)^{l_2,l_1} :: \|\lambda z.\ \ldots\|^{l_2,l_2}\ \bullet, & l_2, & \|1\|^{l_1,l_2}\rangle\rangle \\
\longrightarrow & \ldots & & \\
\longrightarrow & \langle \bullet, & l_0, & \|2\|^{l_1,l_3,l_2,l_1}\rangle
\end{array}
$$

Figure 7 Abbreviated evaluation trace of e_0 from Figure 6, omitting repeated colors for brevity and clarity.

4.1 Program traces

The trace of a program's evaluation is the sequence of currently executing components; such traces are paths in the value graph. To this precise, let $\mathsf{trace}\,(\langle C_1, l_1, e_1\rangle \longrightarrow \langle C_2, l_2, e_2\rangle \longrightarrow \ldots \longrightarrow \langle C_n, l_n, e_n\rangle)$ be defined as $l_1, l_2, \ldots, l_n$ – the payoff of our abstract machine (Section 2.2). For our purposes, a path in a graph is a list of nodes $l_1, l_2, \ldots, l_n$ such that the edge $\{l_{i-1} \leftrightarrow l_i\}$ is in the graph.

▶ **Lemma 4.1** (Traces are paths). *If* $\vdash \langle C, l, e\rangle : l''$, *then* $\mathsf{trace}\,(\langle C, l, e\rangle \longrightarrow^* \langle C', l', e'\rangle)$ *is a path in* $\mathbf{ig}(\langle C, l, e\rangle)$.

Proof. By induction on the length of the trace, observing that component changes must be edges in the interface graph (Lemma A.4) and reducing a term yields a subgraph (Lemma A.3). ◀

4.2 Value colorings

As program e evaluates, the values in e will gain colors – the colors on these values form paths in e's interface graph. We are particularly interested in source programs (with no coloring at all), but this property holds whenever $\mathbf{vg}(e) \subseteq \mathbf{ig}(e)$.

▶ **Lemma 4.2** (Value colorings are paths). *If $\vdash st : l''$ and $\mathbf{vg}(st) \subseteq \mathbf{ig}(st)$ and $st \longrightarrow^* st'$, then for any value $v = \|u\|^{l_1,\ldots,l_n}$ in st', the path $l_1, \ldots, l_n$ is in $\mathbf{ig}(st)$.*

Proof. By induction on the length of the evaluation, strengthening the induction hypothesis to show also that $\mathbf{vg}(st')$ is a subgraph of $\mathbf{ig}(st)$. If $st \longrightarrow^* st'$, then $\mathbf{vg}(st') \subseteq \mathbf{vg}(st) \cup \mathbf{ig}(st)$ (Lemma A.3) in line with our strengthened IH. ◀

4.3 Discussion

The properties we show here are intuitive and simple – arguably too simple. Preservation of well coloring (Lemma 2.2) and our coloring rules guarantee that colors are not trivial and our semantics updates colors correctly when traversing components. But who is to say that the program has interesting colors at all? Values in a "blob" program with a single module indeed trace a path in the interface graph, but that path is nothing more than a trivial self loop. There's nothing we can do about program structure: if the program has few or no modules, the program's graphs will be meager and paths will tell you little or nothing.

We have not applied our generalized analysis to show any more interesting property. It is not hard to recover the core of Dimoulas et al.'s proof: replace any $\mathsf{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c) \rfloor, e)$ with a dynamic check that $e_c\ e$ yields true. Recovering more – dependent contract checking in the variety of styles – would require altering the semantics to closer match theirs. We also envision applications in object capability or other access control enforcement mechanisms. Our graph-theoretic theorems could guide an auditing process, guiding developers through the (syntactically non-obvious) series of interfaces that a value might traverse at runtime. Inspecting interfaces in the order that values will traverse them should help developers avoid classic and common issues with *confused deputies*, i.e., proxies that mediate access but can be tricked to escalate or misuse privileges.

5 Scope graphs (and other related work)

The Clam calculus is directly based on Dimoulas et al.'s work on contracts [2, 3], but there is a long line of research on components, containment, and interfaces in general and in the lambda calculus in particular.

On the theoretical side, the colors in the Clam calculus are similar to the labels in labeled lambda calculus [5]. Their labels differ from ours in a few ways: they are one-sided, while our interfaces and colored expressions are double-sided; they use over- and underlining to distinguish the parts of a β reduct; their semantics has no notion of a "current component"; they allow unlabeled values; and they work in a pure lambda calculus setting, without constants, operations, or conditional control flow. In principle, it might be possible to fit the Clam calculus into the labeled lambda calculus framework – though what advantage this would bring is unclear. The purposes are different, too: our labels are for reasoning about components, but Lévy is more interested in subtle properties of lambda calculus evaluation.

On the practical side, the informal notion of an interface or dependency graph is as old as modularity itself. Components have long been used to reason about "locality" or "containment" – for famous examples, see Morris [7], Dennis and Van Horn [1], or Miller [6]. Zdancewic et al. provide a more closely related notion of ownership [14], cited as direct inspiration by Dimoulas et al. [3].

Finally, Eelco Visser's scope graphs [8] are not dissimilar to the Clam calculus: Visser's scope graphs offer much more complete static [12] and dynamic [10, 13] accounts of binding and the flow of values in a program than either the original analysis or our extension – though scope graphs don't (by default) track the information in Clam's colors.

Compared to the Clam calculus's ifc nodes and colored expressions, scope graphs support a wide range of binding forms, including much more realistic models of modules ([8], Section 2.4). Scope graphs' resolution paths don't quite match the semantics of our colored expressions – but we can imagine extending resolution paths to track detailed path information. Such "colored resolution paths" would give a cleaner dynamic semantics [10], admitting clearly specified and meaningful static analyses [12, 11] – though they would not (without substantial work) track paths in the interface graph in quite the same way. The Clam calculus's model is not tuned for implementation (e.g., rule IfcEndLam in Figure 3), but scope graphs readily admit prototype-level implementations [10] or better [13].

6 Conclusion

The Clam calculus refines Dimoulas et al.'s imprecise dynamic analysis, relating interfaces and value to straightforward graph-theoretic concepts. We offer the Clam calculus as a reusable generalization of Dimoulas et al.'s dynamic analysis – a starting point for reasoning about modularity and runtime control.

References

1 Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9(3), March 1966. doi:10.1145/365230.365252.

2 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*. ACM, 2011. URL: http://www.ccs.neu.edu/home/chrdimo/pubs/popl11-dfff.pdf.

3 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, volume 7211 of *LNCS*. Springer, 2012. URL: http://www.ccs.neu.edu/home/chrdimo/pubs/esop12-dthf.pdf.

4 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*. ACM, 2002. doi:10.1145/581478.581484.

5 Jean-Jacques Lévy. Tracking redexes in the lambda calculus, 2022. In submission. URL: http://pauillac.inria.fr/~levy/pubs/22trackredex.pdf.

6 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. URL: http://www.erights.org/talks/thesis/markm-thesis.pdf.

7 James H. Morris. Protection in programming languages. *CACM*, 16(1), January 1973. doi:10.1145/361932.361937.

8 Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *ESOP*, volume 9032 of *LNCS*. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

9 G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977. doi:10.1016/0304-3975(77)90044-5.

10 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In *ECOOP*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.20.

11 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *PACMPL*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

12 Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*. ACM, 2016. `doi:10.1145/2847538.2847543`.

13 Vlad Vergu, Andrew Tolmach, and Eelco Visser. Scopes and Frames Improve Meta-Interpreter Specialization. In *ECOOP*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2019.4`.

14 Steve Zdancewic, Dan Grossman, and J. Gregory Morrisett. Principals in programming languages: A syntactic proof technique. In *ICFP*. ACM, 1999. URL: `http://www.eecs.harvard.edu/~greg/papers/pipl.pdf`.

A Proofs

▶ **Definition A.1** (Subgraphs). *A graph gr_1 is a subgraph of a graph gr_2 if the nodes of gr_1 are a subset of the nodes of gr_2 and if an edge is in gr_1 then it is in gr_2.*

▶ **Lemma A.2** (Substitution yields subgraphs).

1. $\mathbf{ig}(e\{v/x\})$ *is a subgraph of* $\mathbf{ig}(e) \cup \mathbf{ig}(v)$.
2. $\mathbf{vg}(e\{v/x\})$ *is a subgraph of* $\mathbf{vg}(e) \cup \mathbf{vg}(v)$.

Proof. By induction on e. The graphs are equal if x occurs free in e; if x doesn't occur, then the resulting graph is just e's. ◀

▶ **Lemma A.3** (Reduction yields subgraphs). *If* $\vdash st_1 : l$ *and* $st_1 \longrightarrow st_2$ *then*

1. $\mathbf{ig}(st_2)$ *is a subgraph of* $\mathbf{ig}(st_1)$, *and*
2. $\mathbf{vg}(st_2)$ *is a subgraph of* $\mathbf{ig}(st_1) \cup \mathbf{vg}(st_1)$.

Proof. By case analysis on the reduction step taken. ◀

▶ **Lemma A.4** (Component changes are edges). *If* $\langle C, l, e\rangle \longrightarrow \langle C, l', e'\rangle$, *then* $\{l \leftrightarrow l'\}$ *is in* $\mathbf{ig}(\langle C, l, e\rangle) \cup \mathbf{vg}(\langle C, l, e\rangle)$.

Proof. By case analysis on the step taken. ◀

▶ **Lemma A.5** (Subpaths). *If $l_1, \ldots, l_n$ is a path in a graph gr, then it is also a path in all supergraphs of gr.*

Proof. Given a supergraph gr', by induction on the length of the path. In the inductive case, since gr is a subgraph of gr', every edge in the path in gr must be present in the gr'. ◀

Conf Researchr: A Domain-Specific Content Management System for Managing Large Conference Websites

Danny M. Groenewegen ✉
Delft University of Technology, The Netherlands

Elmer van Chastelet ✉
Delft University of Technology, The Netherlands

Max M. de Krieger ✉
Delft University of Technology, The Netherlands

Daniel A. A. Pelsmaeker ✉
Delft University of Technology, The Netherlands

Craig Anslow ✉
Victoria University of Wellington, New Zealand

Abstract

Conferences are great opportunities for sharing research, debating solutions, and networking. For the organizing committee there is a considerable deal of complexity and effort required to provide attendees and organizers with ways to find and manage programs, sessions, papers, tracks, talks, and authors. Eelco Visser found an opportunity to provide an integrated solution to these problems by designing the Conf Researchr conference management system in 2014 using our own domain-specific web programming language WebDSL. In this paper, we highlight the impact Eelco had on conference management, and how Conf Researchr evolved to become the platform of choice for hosting over 900 conference and workshop editions in SIGPLAN and SIGSOFT, among other areas of computer science research.

2012 ACM Subject Classification Information systems → Social networks

Keywords and phrases Conf Researchr, conferences, WebDSL, Eelco Visser

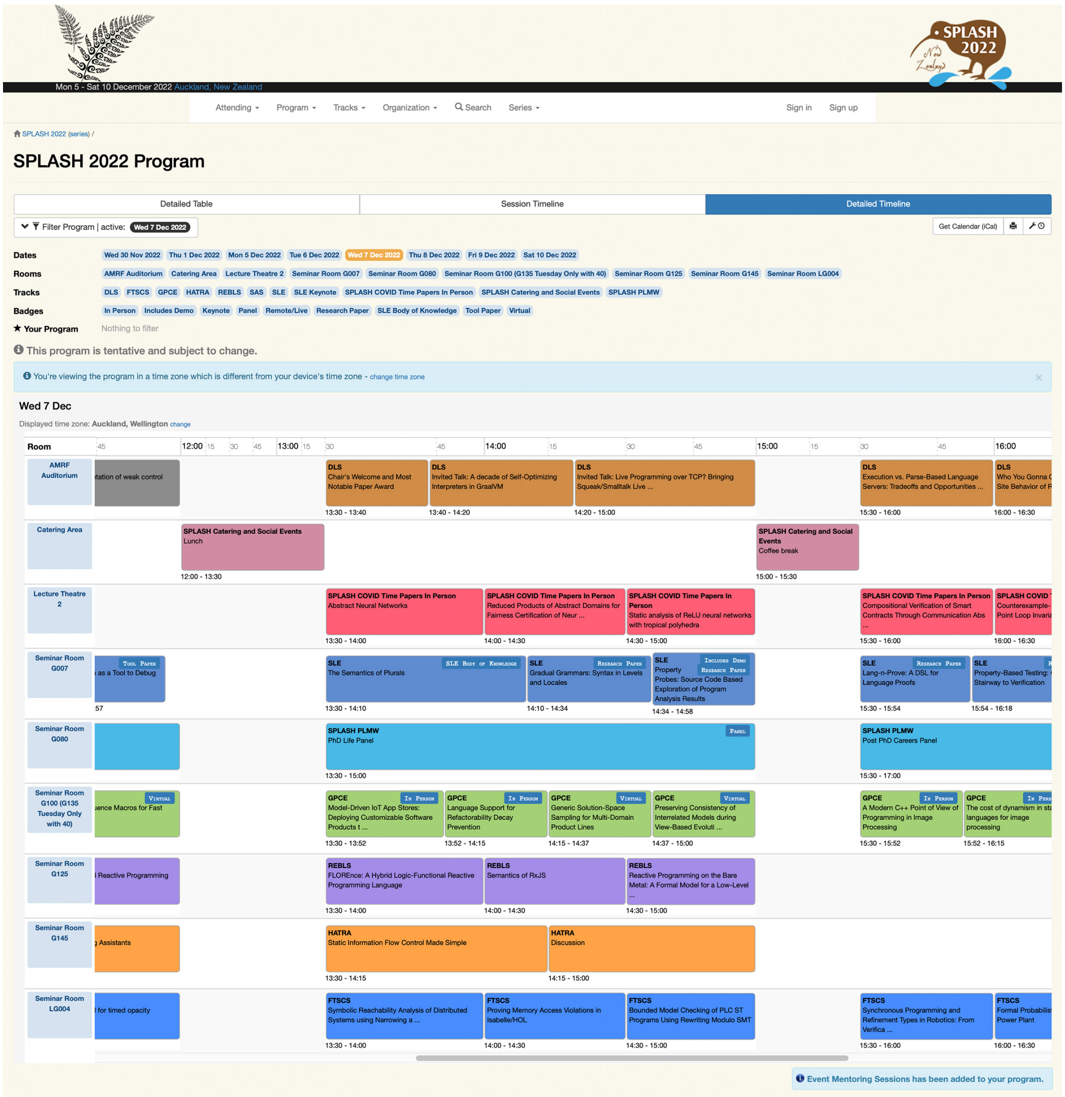
Digital Object Identifier 10.4230/OASIcs.EVCS.2023.12

Acknowledgements We want to thank Eelco Visser for his guidance and vision.

1 Introduction

Federated conferences such as SPLASH and ICSE are complex organizations, consisting of many parts organized by volunteers. A main conference often has multiple tracks and co-located conferences and workshops. For example, the SPLASH conference consists of various tracks including OOPSLA, Onward!, invited speakers, tutorials, and panel discussions. Additionally, there are co-located conferences such as SLE, GPCE, and DLS, and workshops such as REBLS and LIVE, often scheduled in the days leading up to the main conference. Each of these parts has its own steering, organizing, and program committees that select and schedule the keynotes and presentations based on accepted papers or abstracts. These presentations should be put together into a program that the attendees of the conference can use to decide what to attend. While there has been considerable attention for organizing and automating the paper submission and review process (supported by tools such as EasyChair[1],

[1] https://easychair.org/



Mon 5 - Sat 10 December 2022 Auckland, New Zealand

SPLASH 2022
New Zealand
Attending ▾ Program ▾ Tracks ▾ Organization ▾ Search Series ▾ Sign in Sign up
SPLASH 2022 (series) /

SPLASH 2022 Program

Detailed Table | Session Timeline | Detailed Timeline
Filter Program | active: Wed 7 Dec 2022
Get Calendar (iCal)

Dates: Wed 30 Nov 2022, Thu 1 Dec 2022, Mon 5 Dec 2022, Tue 6 Dec 2022, Wed 7 Dec 2022, Thu 8 Dec 2022, Fri 9 Dec 2022, Sat 10 Dec 2022

Rooms: AMRF Auditorium, Catering Area, Lecture Theatre 2, Seminar Room G007, Seminar Room G080, Seminar Room G100 (G135 Tuesday Only with 40), Seminar Room G125, Seminar Room G145, Seminar Room LG004

Tracks: DLS, FTSCS, GPCE, HATRA, REBLS, SAS, SLE, SLE Keynote, SPLASH COVID Time Papers In Person, SPLASH Catering and Social Events, SPLASH PLMW

Badges: In Person, Includes Demo, Keynote, Panel, Remote/Live, Research Paper, SLE Body of Knowledge, Tool Paper, Virtual

★ Your Program: Nothing to filter

This program is tentative and subject to change.

You're viewing the program in a time zone which is different from your device's time zone - change time zone

Wed 7 Dec

Displayed time zone: **Auckland, Wellington** change

Room 45 12:00 15 30 45 13:00 15 30 45 14:00 15 30 45 15:00 15 30 45 16:00

AMRF Auditorium
- tation of weak control
- DLS Chair's Welcome and Most Notable Paper Award 13:30 - 13:40
- DLS Invited Talk: A decade of Self-Optimizing Interpreters in GraalVM 13:40 - 14:20
- DLS Invited Talk: Live Programming over TCP? Bringing Squeak/Smalltalk Live ... 14:20 - 15:00
- DLS Execution vs. Parse-Based Language Servers: Tradeoffs and Opportunities ... 15:30 - 16:00
- DLS Who You Gonna C Site Behavior of F 16:00 - 16:30

Catering Area
- SPLASH Catering and Social Events Lunch 12:00 - 13:30
- SPLASH Catering and Social Events Coffee break 15:00 - 15:30

Lecture Theatre 2
- SPLASH COVID Time Papers In Person Abstract Neural Networks 13:30 - 14:00
- SPLASH COVID Time Papers In Person Reduced Products of Abstract Domains for Fairness Certification of Neur ... 14:00 - 14:30
- SPLASH COVID Time Papers In Person Static analysis of ReLU neural networks with tropical polyhedra 14:30 - 15:00
- SPLASH COVID Time Papers In Person Compositional Verification of Smart Contracts Through Communication Abs ... 15:30 - 16:00
- SPLASH COVID Counterexample- Point Loop Invaria 16:00 - 16:30

Seminar Room G007
- Tool Paper as a Tool to Debug :57
- SLE SLE Body of Knowledge The Semantics of Plurals 13:30 - 14:10
- SLE Research Paper Gradual Grammars: Syntax in Levels and Locales 14:10 - 14:34
- SLE Includes Demo Research Paper Property Probes: Source Code Based Exploration of Program Analysis Results 14:34 - 14:58
- SLE Research Paper Lang-n-Prove: A DSL for Language Proofs 15:30 - 15:54
- SLE Property-Based Testing: Stairway to Verification 15:54 - 16:18

Seminar Room G080
- SPLASH PLMW Panel PhD Life Panel 13:30 - 15:00
- SPLASH PLMW Post PhD Careers Panel 15:30 - 17:00

Seminar Room G100 (G135 Tuesday Only with 40)
- Virtual uence Macros for Fast
- GPCE In Person Model-Driven IoT App Stores: Deploying Customizable Software Products t ... 13:30 - 13:52
- GPCE In Person Language Support for Refactorability Decay Prevention 13:52 - 14:15
- GPCE Virtual Generic Solution-Space Sampling for Multi-Domain Product Lines 14:15 - 14:37
- GPCE Virtual Preserving Consistency of Interrelated Models during View-Based Evoluti ... 14:37 - 15:00
- GPCE In Person A Modern C++ Point of View of Programming in Image Processing 15:30 - 15:52
- GPCE In Pers The cost of dynamism in sta languages for image processing 15:52 - 16:15

Seminar Room G125
- Reactive Programming
- REBLS FLOREnce: A Hybrid Logic-Functional Reactive Programming Language 13:30 - 14:00
- REBLS Semantics of RxJS 14:00 - 14:30
- REBLS Reactive Programming on the Bare Metal: A Formal Model for a Low-Level ... 14:30 - 15:00

Seminar Room G145
- Assistants
- HATRA Static Information Flow Control Made Simple 13:30 - 14:15
- HATRA Discussion 14:15 - 15:00

Seminar Room LG004
- for timed opacity
- FTSCS Symbolic Reachability Analysis of Distributed Systems using Narrowing a ... 13:30 - 14:00
- FTSCS Proving Memory Access Violations in Isabelle/HOL 14:00 - 14:30
- FTSCS Bounded Model Checking of PLC ST Programs Using Rewriting Modulo SMT 14:30 - 15:00
- FTSCS Synchronous Programming and Refinement Types in Robotics: From Verifica ... 15:30 - 16:00
- FTSCS Formal Probabilis Power Plant 16:00 - 16:30

Event Mentoring Sessions has been added to your program.

Figure 1 A screenshot of the detailed timeline overview of the SPLASH'22 conference, the eighth edition of SPLASH that uses the Conf platform, with over 450 in-person and virtual attendees.

HotCRP[2], and Conference Publishing Consultancy[3]), developing the website for such a conference also requires considerable effort and input from many people over an extended period of time. This process is often reinvented for each yearly edition of a conference, by a fresh team of volunteers, using software that provides little to no support for the domain. As a result, information for many tracks is often not integrated in the main website, and each workshop maintains their own website from which the attendees have to harvest calls for papers and programs.

Eelco Visser came up with the idea for the Conf Researchr [2] conference management system to provide an integrated solution for such problems. He promoted the use of Conf Researchr actively in his network of conference organizers, always ready to provide a demo

[2] https://hotcrp.com/
[3] https://www.conference-publishing.com/

on his laptop. His enthusiasm drove the continuous development and adoption of the system. Conf Researchr has been developed by the Academic Workflow Engineering team that Eelco formed at Delft University of Technology. It uses the WebDSL programming language, leveraging Eelco's earlier research into domain-specific languages for web programming. In this paper, we report on experiences of operating and evolving Conf Researchr.

2 Feature Evolution

Conf Researchr is implemented as a *multi-tenant* web application, hosting the web sites for multiple conference series and editions in a single application. Multi-tenancy is realized by using the domain name or URL path to denote the viewing context for each page, which is typically the conference or workshop itself, but may also be the hosting conference in case of a co-located event. The viewing context is used at various places in the implementation that are responsible for rendering the pages, and is designed to create a unique look-and-feel for each tenant while enjoying the recognizable structure of Conf Researchr. It determines how navigation menus are constructed, whether or not a custom style sheet should be included and which, possibly role-dependent, links a visitor has access to. As these tenants often operate under distinct domain names specific to each conference, requesting and renewing SSL certificates for each edition's domain name has been automated on the server using Let's Encrypt[4].

A central feature of Conf Researchr is managing and viewing a conference program. Conference attendees can discover the conference program through extensive filtering options, and add events to their individual program to compose a personal schedule to use during the conference. Figure 1 shows an example of the timeline program view for SPLASH 2022.

Conf Researchr is a major timesaver for conference organizers, who, instead of painfully putting together a program in a spreadsheet and converting it to HTML, are provided with a system specifically geared toward this purpose. The first step in configuring a program is to create session slots that indicate the time slots available for presentations and other sessions. Then, conference tracks are assigned to session slots to indicate the days and times in which the track is running. Events that fit into the track sessions are selected from the imported list of accepted papers and other submissions. These events are evenly spread across the session time by default, and the times can be adjusted if necessary. Author profiles are automatically matched and linked based on their email address, and otherwise are easy to look up and configure by the organizers.

Since the initial implementation that was first used by SPLASH 2014, Conf Researchr evolved through requested quality of life improvements and additional features from conference organisers to streamline the processes. Examples of these additions include: importing events and authors from additional paper submission systems, copying content from previous editions, conflict detection while scheduling sessions, providing a data API for mobile applications, shared schedules between tracks, and a workshop proposal system.

Conference contributors have a customizable profile page that features their current affiliation, a small biography, and the contributions and roles within a conference. At the time of the conference, a snapshot is created of this conference-specific profile page, to give an insight into the activities and interests of the contributor at that point in time. The general profile of a user gives an overview of all their past contributions and roles, and becomes more accurate as an academic portfolio when more conferences in a field use the system.

[4] https://letsencrypt.org/

Next to profile pages, Conf Researchr automatically creates a *people index* of everyone that is involved on the conference level, or within a track. With filtering and advanced search in place, the people index gives an opportunity to interactively discover the contributors, committee members and session chairs.

For providing a historical perspective and easy retrieval of passed events, it is essential to keep a conference website online after it is finished. Thanks to the multi-tenant setup of our system, we can conveniently keep hosting previous editions of conferences. Some conference organizers even retroactively set up editions in Conf Researchr for past conferences that no longer had a working website. Conf Researchr also provides a download of a static backup of the conference website, mainly for continuity purposes but also to serve as emergency backup in case the Conf Researchr services would ever be down during a conference.

The COVID-19 pandemic greatly impacted the organization of conferences. We adapted Conf Researchr to tackle the challenges of arranging remote and hybrid conferences. Features created for online conferences include mirrored sessions for different time zones, remote meeting links for participants, and embedding pre-recorded presentations in conference events.

3 Implementation Evolution

Conf Researchr is developed using WebDSL [3, 1], a domain-specific programming language for the development of full-stack web applications that integrates sub-languages for data persistence, user interfaces, access control, data validation, and full text search. WebDSL enables rapid prototyping of web applications and allows for customization to further improve the performance and user experience.

While the WebDSL language has been gradually extended and improved over time, some features were added to WebDSL primarily to support Conf Researchr. This includes the ability to customize URL routing instead of depending on an automatic URL scheme, to support multi-tenancy on different domain names. While the standard pattern for WebDSL page URLs consisted of a page name followed by slash-separated URL representations of page arguments, for Conf Researchr the domain name is also used to determine one of these arguments, namely the conference viewing context.

Another added feature was an automatic page cache in WebDSL, to optimize applications with relatively stable content including Conf Researchr. Furthermore, features were added to WebDSL to shorten several code fragments, such as static code templates to create a library of input components with consistent HTML element wrappers and styling.

The first deployed version of Conf Researchr in October 2014 was around 8,500 lines of WebDSL code. The current implementation consists of roughly 30,000 lines of WebDSL code. Of this codebase, approximately 60% is user interface code in the form of WebDSL pages and templates (displaying HTML with entity data and handling user interface actions), 30% is used for persisted entities and their functions (data objects with methods), and the remaining 10% are other definitions such as access control rules, static code templates, web services for the data API, and search configurations.

Database migrations have been automatic, as these have only consisted of adding elements to the data model. After adding new entities and entity properties to existing WebDSL applications, new tables and columns are created automatically when a new application version is deployed. For large tables, adding a column can take a while, in these cases we sometimes performed the schema change manually before deploying the new version to avoid downtime.

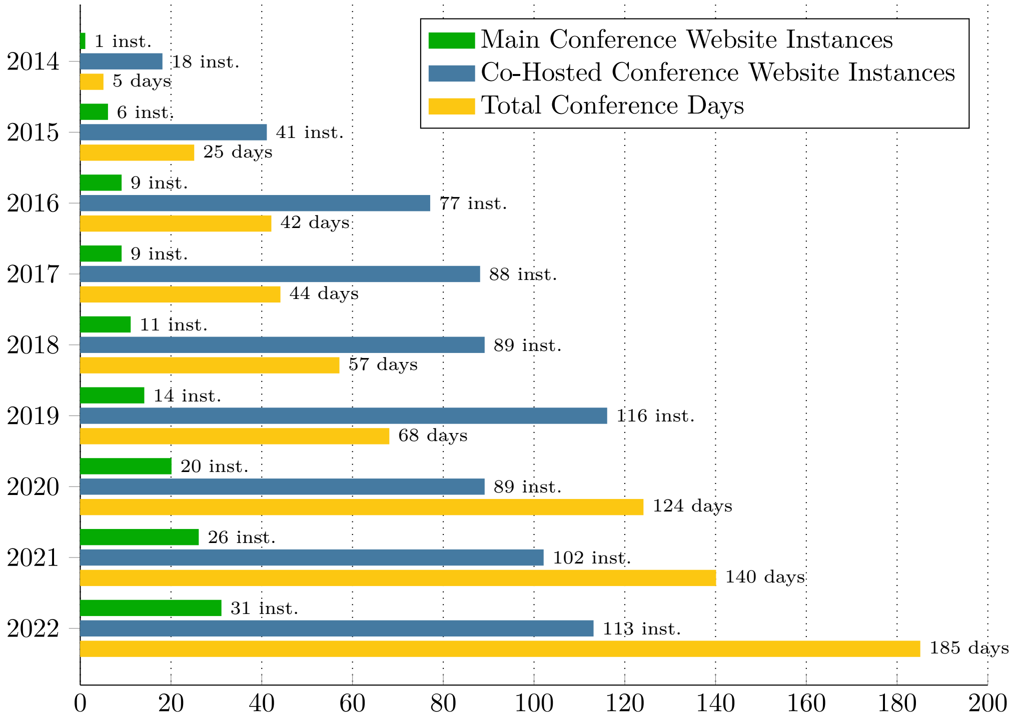


Figure 2 A breakdown showing the number of Conf Researchr website instances for main conferences and co-hosted conferences, and the total number of conference days for each year since the application's inception.

4 Usage

Conf Researchr was first used for SPLASH 2014. Eelco managed to secure contracts that expanded the usage of the application to include all other SIGPLAN conferences, and later SIGSOFT. In addition, many new conference organizers found the Conf Researchr platform through attending other conferences or based on positive experiences from other organizers. At the time of writing, the system has over 24,000 user accounts, and hosts over 900 conference and workshop editions. Figure 2 shows the progression of Conf Researchr usage since its inception.

5 Conclusion

Eelco Visser saw opportunities to improve the status quo with respect to conference websites, that were often created manually for each conference edition. The result of this is the Conf Researchr application, which was first deployed as part of SPLASH 2014. Since then it has been used by hundreds of conference editions and visited by tens of thousands of attendees. The design of the application was directed by Eelco based on his experience as both a recurring conference attendee and committee member. Conf Researchr is still actively being developed at the Delft University of Technology by Eelco's Academic Workflow Engineering team.

References

1 Danny M. Groenewegen, Elmer van Chastelet, and Eelco Visser. Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language. In Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix, editors, *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*, pages 77–83. ACM, 2020. `doi:10.1145/3397537.3397553`.

2 Elmer van Chastelet, Eelco Visser, and Craig Anslow. Conf.Researchr.Org: towards a domain-specific content management system for managing large conference websites. In *OOPSLA*, pages 50–51, 2015. `doi:10.1145/2814189.2817270`.

3 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007. `doi:10.1007/978-3-540-88643-3_7`.

Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows

Danny M. Groenewegen
Delft University of Technology, The Netherlands

Elmer van Chastelet
Delft University of Technology, The Netherlands

Max M. de Krieger
Delft University of Technology, The Netherlands

Daniel A. A. Pelsmaeker
Delft University of Technology, The Netherlands

Abstract

SDF, Stratego and Spoofax provide a platform for development of domain-specific programming languages. On this platform, the WebDSL project started out as a case study in language engineering, and grew into a reliable tool for rapid prototyping and continuous development of web applications. Our team led by Eelco Visser develops and operates several web applications to support academic workflows. EvaTool governs the process of course quality control, importing questionnaire data, and providing lecturers and education directors with a platform to discuss and agree on improvements. WebLab is an online learning management system with a focus on programming education, with support for lab work and digital exams, used by over 40 courses. Conf Researchr is a domain-specific content management system for creating and hosting integrated websites for conferences with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences. MyStudyPlanning is an application for composition of individual study programs by students and verification of those programs by the exam board, used by multiple faculties at the Delft University of Technology. These tools served as practical case studies for applying the research, and ensure the continued development of the underlying platform.

2012 ACM Subject Classification Applied computing → Education

Keywords and phrases WebDSL, WebLab, Spoofax, Eelco Visser, education

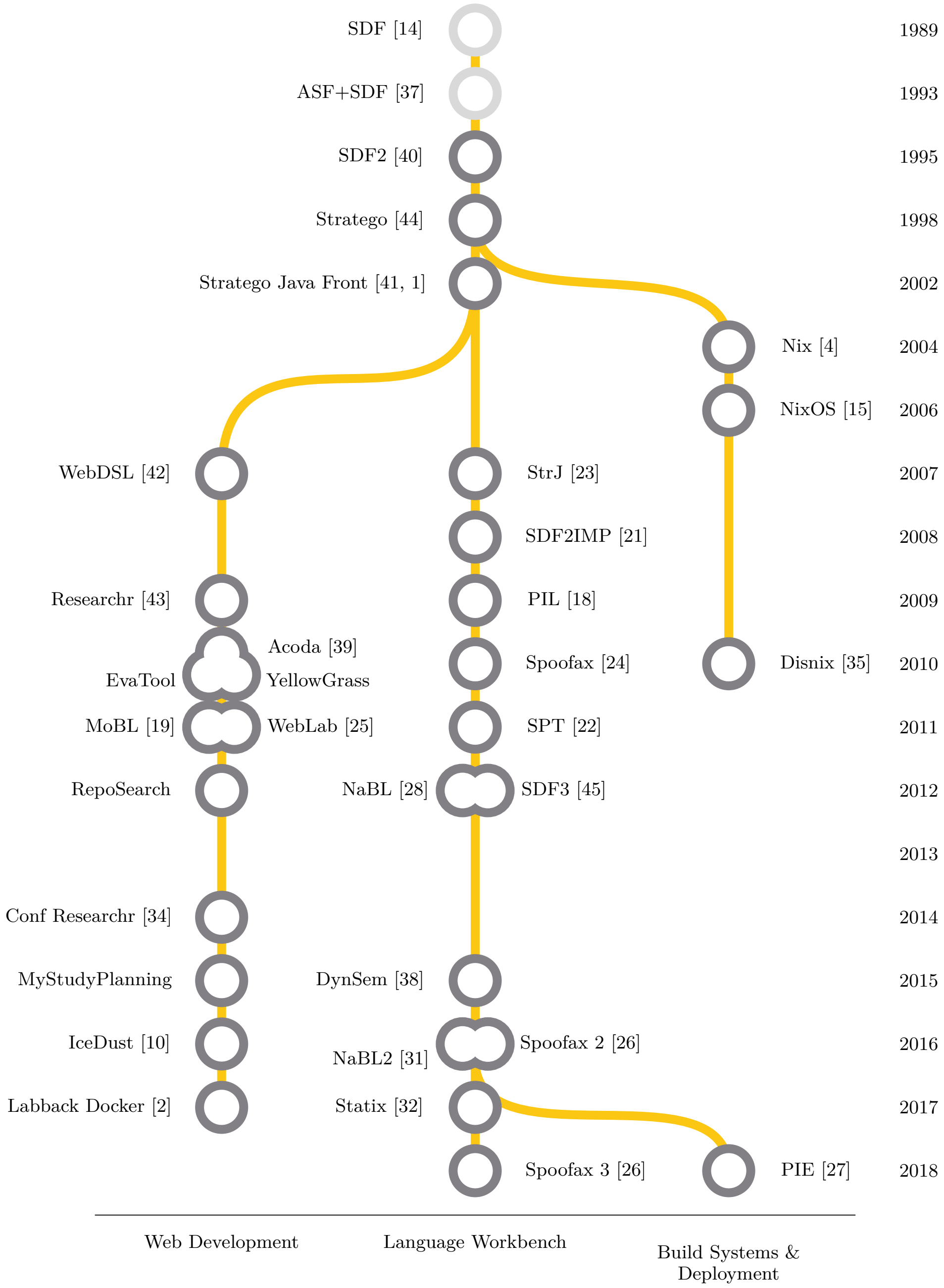
Digital Object Identifier 10.4230/OASIcs.EVCS.2023.13

Acknowledgements We want to thank Eelco Visser for his guidance and vision.

1 Introduction

Software that is produced as part of academic research often remains in a prototype state. There is tension between investing time in improving the quality of the software and reporting on the software through publications and presentations. By applying the software to practical real-world case studies, both the quality of the software improves by discovering and solving issues, and the reporting becomes stronger through meaningful evaluation. Applying research software in practice also generates new questions to explore. Eelco Visser's vision for language design and engineering involves a recurring pattern of building upon software contributions from previous work, and applying the research ideas to new practical case studies. This research expanded from the fields of language engineering into build and deployment tools and web application engineering. Eelco brought together the people that worked on the projects, primarily as part of Master and PhD thesis projects. Figure 1 shows a timeline of notable projects by Eelco and his students.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 13; pp. 13:1–13:11
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** A timeline of notable projects by Eelco Visser and his students. Eelco improved the Syntax Definition Formalism [14] (SDF) as part of his PhD on SDF2 [40]. The Stratego language [44] was designed as an improvement on ASF+SDF [37], allowing programmers complete control of traversal strategies for term rewriting. Bootstrapping the Stratego compiler initiated research into better build systems with Nix [4], which was incorporated as the package manager for the Linux distribution NixOS [15]. WebDSL [42] grew from the wish to improve processes in academia, and served as the largest case study in research that led to the full-fledged language workbench Spoofax [24] to generate IDEs. At the time of writing, several real-world web applications with thousands of users, built using WebDSL and Spoofax, are running on NixOS servers.

Eelco had a strong wish for software to automate the mundane and repetitive parts of working in academia, and therefore Eelco advocated to form a dedicated development team within the department, tasked with writing and maintaining such tools for managing the academic workflow. This was a tough sell, since the university management wanted to avoid depending on in-house developed tools, rather outsourcing both the effort and the responsibility to third-party software companies. When a top-down approach did not pan out, Eelco decided to improve the academic workflow from the bottom-up, starting with automations that would help him in his own courses and his dealings with the students he was supervising, and one approach was to create web applications. Eelco's initial prototype, used to gather requirements for the design of WebDSL, was a web application for tracking master student projects and progress, to replace a previous generic wiki application that lacked any domain knowledge. However, it is difficult to find the resources for maintaining the WebDSL language and applications merely through PhD projects. As a result, the starting point for more widely used applications came when Eelco formed the Academic Workflow Engineering team around 2013, initially consisting of Danny Groenewegen and Elmer van Chastelet, and later joined by Daniel Pelsmaeker and Max de Krieger. We became a permanent team of software developers to continue WebDSL development, and develop new web applications based on real case studies with clients. Initially, most web applications were used just within the Programming Languages group that Eelco led, but thanks to Eelco promoting the applications, they gradually started being adopted in other groups within the Computer Science faculty, then outside the faculty and even outside the university.

Section 2 highlights the main benefits of WebDSL for creating web information systems, and explains how it was built upon previous research by Eelco Visser. The Researchr application (Section 3), an early WebDSL application, assists paper writing by indexing computer science papers, managing bibliographies, and generating required BibTeX files. EvaTool (Section 4) governs the process of course quality control, importing questionnaire data, and providing lecturers and education directors with a platform to discuss and agree on improvements. The WebLab online learning management system (Section 5) has a focus on programming education (students make programming assignments in the browser), with support for lab work and digital exams. WebLab is currently used by over 40 courses, some even outside Delft University of Technology. Conf Researchr (Section 6) is a domain-specific content management system for creating and hosting integrated websites for conferences with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences. Conf Researchr has been used to serve thousands of attendees at over a hundred editions of dozens of (co-located) conferences and workshops. MyStudyPlanning (Section 7) is an application for composition of individual study plans by students and verification of those plans by the exam board, used by multiple faculties at Delft University of Technology. Finally, we wrap up with the conclusion in Section 8.

2 WebDSL

Web programming suffers from abstraction issues, as web frameworks are developed on top of general-purpose programming languages that were not specifically designed for web programming. As a result, static verification of web programming concepts is limited by the strength of the type checker of the programming language, sometimes augmented with separate linting tools built into IDEs [16]. Security should also be taken into account in the design of web programming languages, otherwise regular programming language features become pitfalls that result in web applications with vulnerabilities. For example, using a simple string interpolation feature for concatenation of SQL queries easily leads to injection vulnerabilities.

The WebDSL project [42, 5] takes a language engineering approach to improving web programming. WebDSL[1] is a linguistically integrated domain-specific language (DSL) for web programming that combines abstractions for web programming concerns covering transparent persistence, user interfaces, data validation [7], access control [6], and internal site search [33]. Sublanguages for the various concerns are integrated into WebDSL through static verification to prevent inconsistencies, with immediate feedback in the IDE and error messages in terms of domain concepts. The user interface sublanguage of WebDSL integrates automatic data binding, provides safety from data tampering, prevents input identifier mismatch in action handlers, enables safe composition of input components, enforces cross-site request forgery protection, allows expressive data validation, and supports partial page updates without explicit JavaScript or DOM manipulation. The access control sublanguage allows various policies to be expressed with simple constraints. Enforcement of access control is done by automatically weaving checks into the application code. The explicit declaration of access control though language elements allow assumptions to be made about the related parts in the application, such as hiding inaccessible navigation links. WebDSL is designed and most suitable for building full-stack web information systems, applications with a rich data model, data entry with validation rules, and a multi-user access control policy.

WebDSL development was started by Eelco Visser as a case study for using SDF and Stratego to build real-world programming language compilers [42]. His earlier research on language embedding, together with Martin Bravenboer, resulted in libraries for embedding Java syntax in Stratego code [1], which provided convenient programming tools for code generation. Eelco, together with PhD students Zef Hemel, Lennart Kats, and Danny Groenewegen, used the WebDSL project to discover how to structure compilers and which programming patterns to use in a system that provides rewriting rules with concrete syntax embedding [17]. In parallel, Lennart Kats worked on Spoofax, which consists of Eclipse IDE support for language definitions written in SDF and Stratego [24], and the Stratego Java compiler backend [23] which expanded on initial work on a Stratego Java interpreter by Eelco's former PhD student Karl Trygve Kalleberg [20]. Eelco observed that name binding rules generated complexity in the rewriting rules of the WebDSL compiler implementation, which inspired research into better abstractions for specifying name binding and static semantics, in particular NaBL [29], NaBL2 [31], and ultimately Statix [32]. WebDSL is still the largest case study for these tools, as shown in the encoding of WebDSL static semantics in SDF3 and Statix by Max de Krieger [3].

Another development aspect of compilers is build systems. The bootstrapping required to build Stratego required better tool support, which was initial inspiration for the Nix project by PhD student Eelco Dolstra [4]. This project grew into the full Linux distribution NixOS[2] based on the Nix principles for reproducible, declarative and reliable system specification. WebDSL application deployment is still done on NixOS servers. One specification describes a complete system configuration, which, for example, contains multiple MySQL and Tomcat installations and an Nginx reverse proxy server.

[1] https://webdsl.org/

[2] https://nixos.org/

3 Researchr

Researchr[3] [43] was one of the first case studies with WebDSL. It is a web application that can search and index computer science publications, and perform bibliography management. At the time, this project was motivated by Eelco's frustration with the DBLP Computer Science Bibliography, a large computer science bibliography website, that would not list papers that were not in the curated list of conferences. Eelco was also convinced that much of the time and effort required to search and use literature was a result of poor tool support and the fact that different independent tools were used for indexing, managing, and classification of publications. Therefore, Eelco wanted Researchr to be the one-stop-shop for finding and categorizing publications, literature review, and managing bibliographies. This requires a big catalog of publications, so Researchr automatically imports all the publications indexed by DBLP, but also allows users to contribute and improve publication metadata.

Although the initial prototype of WebDSL was lacking features for user management, at this time we had developed essential language additions for making complete applications through the exploration of access control [6] and data validation [8]. Researchr was used as a test bed for new developments in the WebDSL compiler, in particular for the renewed implementation of the WebDSL runtime [5]. As the Researchr project grew, so did the demands on WebDSL. For example, the large set of publications in Researchr required an effective and efficient search feature. Implementing such a system with searching entities in the database became too slow and this inspired the development of a dedicated WebDSL language feature for search using Lucene [33]. The project was also a great way to find bugs, memory leaks, and performance problems in WebDSL, and as WebDSL became more stable Eelco worked on Researchr more as a continuous side project, occupying his mind even on holidays.

4 EvaTool

For the performance of courses and evaluating our education at the Delft University of Technology, various statistics are gathered about the courses, as well as feedback from students. These statistics are collected in various different systems across the university, such as EvaSys for surveys, Osiris for grades, and summaries from student feedback groups. Additionally, there is a need to record the instructor response to the course performance, and to record future plans for the course. All this information used to be sent around using ad-hoc emails and was monitored manually in spreadsheets by the quality assurance department. As a result, EvaTool[4] was developed to collect and analyze this information in a single place, and gain better insight into the educational system. The project was initiated by Eelco in 2010 after he had several discussions with colleagues from the quality assurance department. A student who took Eelco's course on Model-Driven Software Development was hired to work on the first prototype of EvaTool. A prototype was created with a basic data model supporting the different steps in the course evaluation process. Various iterations of this prototype were used to tweak functionality and to search for the right workflow and terminology to be valuable in a broader setting with multiple faculties.

EvaTool transformed the ad-hoc nature of course quality control into a directed process of education evaluation, covering the collection and filtering of data, informing relevant people, asking for response and intended improvements, and projecting the evaluation data in the

[3] `https://researchr.org/`

[4] `https://evatool.tudelft.nl/`

various formats suitable for management or students. Over the years, EvaTool gained traction, resulting in other faculties now using the tool to manage their education evaluation process. As these processes differ slightly between faculties, EvaTool was extended with faculty-specific customizations, including customizable roles and optional steps in the evaluation process. The tabular and textual reporting capabilities were extended with a template designer, enabling more visual fact sheets to be generated from the course evaluations.

5 WebLab

WebLab[5] is an online learning environment, which is currently used by courses both inside and outside Delft University of Technology for practice materials, lab assignments, programming exercises, and exams. It started around 2011, while Eelco was teaching Model-Driven Software Development and concluded that it would be much easier to have the students fill out multiple-choice and open questions online than to try to read the various answers on paper. Additionally, if the students had to submit code, Eelco argued that there should be a platform to which they can submit instead of students emailing the course instructor with a ZIP archive of the code. Finally, once the questions and submissions are in an online system, it could also be used to hold the grades for the individual submissions and automatically calculate the overall grade for the student. This inspired the creation of WebLab, which thanks to previous positive experiences was also written in WebDSL.

While Eelco was teaching Concepts of Programming Languages, it became apparent that providing an online editor for student code submissions would not only solve the problem of students having to install development software locally, but also provide an opportunity to perform automatic grading of the code submissions. The initial version started with an email to Danny in December 2011, where Eelco asked how to integrate the Cloud9's Ace editor component in WebDSL code. This simple code editor provided syntax highlighting for the languages in which the course was taught, such as Java and Scala [36]. For programming assignments and automated grading, Vlad Vergu implemented a WebLab *backend* that runs the student's code on the Java Virtual Machine (JVM) in a severely restricted environment. By running a hidden suite of specification tests against the student's code, the test results could be used to determine the grade: the more tests succeed, the better the student is doing. JavaScript support was later added through the JVM's built-in Javascript functionality known as Nashorn, and eventually WebLab also supported C code, which was run using Emscripten and JavaScript on the JVM.

Another course that gave direction to the development of WebLab was Algorithms and Data Structures, which was in essence a stress test for WebLab: previous editions of this course had over 300 students making programming assignments, sometimes all at the same time during an exam (current edition has 600 students). Therefore, it was imperative that WebLab would scale sufficiently to be able to handle all these requests in a timely manner. From this new requirement, the idea of having multiple backends emerged. Each backend would handle and run part of the submissions of the students, and if a backend ended up crashing (for example, due to a student error), a fresh new backend could take its place without impacting the availability of WebLab or the other backends.

As WebLab was growing as a platform and as a case study for WebDSL, it was also useful as a topic for various master and PhD projects. This allowed these projects to extend and improve WebLab and WebDSL, and see their results used in a large application that is actually used in practice. There were bachelor projects that added features such as user groups and SQL support [30].

[5] `https://weblab.tudelft.nl/`

One notable project that generated multiple papers was IceDust [10, 11, 12, 13], part of the PhD work of Daco Harkes [9]. This project resulted directly from the need to streamline the grade calculation and grade dependencies in WebLab, as grade calculations are modelled as *derived values* with complex interactions and dependencies between values (grades) in the data model. In this research he explored the problems associated with derived values and their transitive dependencies, and the performance characteristics of three different strategies for calculating them (on read, on write, or *eventually consistent*).

Eventually, WebLab was used by so many courses that the original JVM-based backends became too restrictive, and were therefore replaced by backends that run isolated Docker containers, as part of another research project [2]. This allows programming assignments for any language for which a Docker image can be created, and it is currently used to provide support for Python, Haskell, Rust, and Agda, among others.

WebLab has been in development over the past decade and gradually gained more and more features, including file submissions, collaborative assignments, and question variants. New features are still being added, such as support for *language servers* using the Language Server Protocol[6] (LSP), learning paths, parameterized questions, and personalized feedback to improve formative testing.

6 Conf Researchr

As an academic, Eelco attended many conferences but noticed that these conferences often had to build a new website for every edition of a conference. He saw an opportunity to help the conference organizers by building a domain-specific content management system (CMS) that would support the needs of conference websites. The conference and many different sub-conferences, workshops, symposia, talks, and other events are coordinated and managed by various organizers and committees, yet should ultimately produce a single conference website that attendees can quickly navigate. With help from Jan Vitek, Eelco secured a contract for funding the development of this system, known as Conf Researchr[7] [34], and the first conference to adopt the system was SPLASH 2014, a large conference with co-hosted subconferences and workshops. Initially, Conf Researchr was meant to be part of Researchr, however, due to time constraints Conf Researchr remained a separate application sharing only the name and the WebDSL implementation language.

Over time, Conf Researchr gained many features for conference organizers, such as support for plenary and blended sessions and timezones. During the COVID-19 pandemic, support was added for virtual conferences with mirrored sessions, such that virtual attendees in other timezones can still catch a talk at an acceptable time. Additionally, conference attendees can create personalized calendars of the talks and events they plan to visit, and export or synchronize this to their personal calendar. As the number of conferences grew, so did the requirements. Integrations with other applications such as HotCRP[8], Conference Publishing Consultancy[9], and EasyChair[10] were added, as well as a simple CMS-mode for workshops and single-track conference websites that do not need all the advanced features. Ultimately, Conf Researchr has supported all SIGPLAN conferences since 2014, ICSE since 2018, and all SIGSOFT conferences since 2020.

[6] https://microsoft.github.io/language-server-protocol/

[7] https://conf.researchr.org/

[8] https://hotcrp.com/

[9] https://www.conference-publishing.com/

[10] https://easychair.org/

7 MyStudyPlanning

At Delft University of Technology, first year Master students are required to plan ahead and select a number of courses to follow as part of their Individual Exam Program (IEP) in order to obtain the required amount of course credits (ECTS). Previously, students filled out a paper form containing this information, which was sent around multiple departments to gather approvals and signatures, and if the student's program changed, the forms had to be adjusted and sent around again. This process could be more streamlined and less error-prone if it were automated, which resulted in MyStudyPlanning.

MyStudyPlanning[11] is a web application, written in WebDSL, that guides the student through the course selection procedure, and even allows early feedback through the use of categories with constraints, automatically selecting mandatory courses, and showing warnings when the student selects invalid combinations. Additionally, both employees registering the program and employees mandated to approve the submitted course selections benefit from MyStudyPlanning by having clear overviews of pending requests, reminders and notifications, and a traceable history of a course selection. Thanks to WebDSL's ability for rapid prototyping, features of MyStudyPlanning could be implemented and demonstrated quickly, which sparked enthusiasm with employees and causes the application to be adopted across multiple faculties within the university. Since its inception in 2016, the MyStudyPlanning application rapidly grew and is now used by Master students in Computer Science, Electrical Engineering, Mathematics, Civil Engineering, Geosciences, and Aerospace Engineering.

8 Conclusion

As part of the research of Eelco and his Programming Languages group, there was a need for thorough case studies to show the advantages of the programming languages and techniques that were being developed. At the same time, Eelco encountered limitations in his work as an educator and researcher that could be mitigated by applying the developed DSLs and tools in these practical settings. These case studies therefore served two purposes: as practical case studies, and to help solve actual problems that he experienced in education and academia.

For many of his projects Eelco had a bottom-up approach: first apply the research to a single, ideally practical, case study and gradually expanding into other case studies from there. By identifying common patterns in the applications, it became more clear which boilerplate code to abstract into a DSL and on which areas to focus next. This was especially apparent in the WebDSL language [42], whose case studies grew into several usable applications that ended up being used in universities and academic conferences to solve real-world problems, including Researchr, EvaTool, WebLab, Conf Researchr, and MyStudyPlanning. Additionally, Eelco was excellent at promoting these applications both within and outside Delft University of Technology, which provided funding for the continued development of these applications.

Eelco's vision and method resulted in real-world impact of research in language design and engineering: MyStudyPlanning and EvaTool are used by many faculties across the Delft University of Technology. WebLab is currently used by more than 40 courses, both from Delft and other universities. Conf Researchr has been used to serve thousands of attendees at over a hundred editions of dozens of (co-located) conferences and workshops. The impact of these applications, some already over a decade old, keeps growing every year.

[11] https://mystudyplanning.tudelft.nl/

References

1 Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA*, pages 365–383, 2004. `doi:10.1145/1028976.1029007`.

2 Bram Crielaard, Chiel Bruin, and Taico Aerts. Native WebLab: Safe execution of native code in WebLab. Bachelor's thesis, Delft University of Technology, 2017.

3 Max M. de Krieger. Modernizing the WebDSL front-end: A case study in SDF3 and Statix. master's thesis. `http://resolver.tudelft.nl/uuid:564b8471-631f-4831-a049-58b187425aed`, 2022.

4 Eelco Dolstra. *The Purely Functional Software Deployment Model.* PhD thesis, Utrecht University, Utrecht, The Netherlands, January 2006.

5 Danny M. Groenewegen, Elmer van Chastelet, and Eelco Visser. Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language. In *Programming*, pages 77–83, 2020. `doi:10.1145/3397537.3397553`.

6 Danny M. Groenewegen and Eelco Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In *ICWE*, pages 175–188, 2008. `doi:10.1109/ICWE.2008.15`.

7 Danny M. Groenewegen and Eelco Visser. Integration of data validation and user interface concerns in a DSL for web applications. In *SLE*, pages 164–173, 2009. `doi:10.1007/978-3-642-12107-4_13`.

8 Danny M. Groenewegen and Eelco Visser. Integration of data validation and user interface concerns in a DSL for web applications. *SoSyM*, 12(1):35–52, 2013. `doi:10.1007/s10270-010-0173-9`.

9 Daco Harkes. *Declarative Specification of Information System Data Models and Business Logic*. PhD thesis, Delft University of Technology, Netherlands, 2019. `doi:10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a`.

10 Daco Harkes, Danny M. Groenewegen, and Eelco Visser. IceDust: Incremental and eventual computation of derived values in persistent object graphs. In *ECOOP*, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.11`.

11 Daco Harkes, Elmer van Chastelet, and Eelco Visser. Migrating business logic to an incremental computing DSL: a case study. In *SLE*, pages 83–96, 2018. `doi:10.1145/3276604.3276617`.

12 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In *SLE*, pages 241–260, 2014. `doi:10.1007/978-3-319-11245-9_14`.

13 Daco Harkes and Eelco Visser. IceDust 2: Derived bidirectional relations and calculation strategy composition. In *ECOOP*, 2017. `doi:10.4230/LIPIcs.ECOOP.2017.14`.

14 Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN*, 24(11):43–75, 1989. `doi:10.1145/71605.71607`.

15 Armijn Hemel. NixOS: the Nix based operating system INF/SCR-05-91. Master's thesis, University of Utrecht, 2006.

16 Zef Hemel, Danny M. Groenewegen, Lennart C. L. Kats, and Eelco Visser. Static consistency checking of web applications with WebDSL. *JSC*, 46(2):150–182, 2011. `doi:10.1016/j.jsc.2010.08.006`.

17 Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402, 2010. `doi:10.1007/s10270-009-0136-1`.

18 Zef Hemel and Eelco Visser. PIL: A platform independent language for retargetable DSLs. In *SLE*, pages 224–243, 2009. `doi:10.1007/978-3-642-12107-4_17`.

19 Zef Hemel and Eelco Visser. Declaratively programming the mobile web with Mobl. In *OOPSLA*, pages 695–712, 2011. `doi:10.1145/2048066.2048121`.

20 Karl Trygve Kalleberg and Eelco Visser. Fusing a transformation language with an open compiler. *Electronic Notes in Theoretical Computer Science*, 203(2):21–36, 2008. `doi:10.1016/j.entcs.2008.03.042`.

21 Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Generating editors for embedded languages. integrating SGLR into IMP. In *LDTA*, April 2008.

22 Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Testing domain-specific languages. In *OOPSLA*, pages 25–26, 2011. doi:10.1145/2048147.2048160.

23 Lennart C. L. Kats and Eelco Visser. Encapsulating software platform logic by aspect-oriented programming: A case study in using aspects for language portability. In *SCAM*, pages 147–156, 2010. doi:10.1109/SCAM.2010.11.

24 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010. doi:10.1145/1869459.1869497.

25 Lennart C. L. Kats, Richard Vogelij, Karl Trygve Kalleberg, and Eelco Visser. Software development environments on the web: a research agenda. In *OOPSLA*, pages 99–116, 2012. doi:10.1145/2384592.2384603.

26 Gabriël Konat. *Language-Parametric Methods for Developing Interactive Programming Systems*. PhD thesis, Delft University of Technology, Netherlands, 2019. doi:10.4233/uuid:03d70c5d-596d-4c8c-92da-0398dd8221cb.

27 Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Scalable incremental building with dynamic task dependencies. In *ASE*, pages 76–86, 2018. doi:10.1145/3238147.3238196.

28 Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *SLE*, pages 311–331, 2012. doi:10.1007/978-3-642-36089-3_18.

29 Gabriël Konat, Vlad A. Vergu, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. The Spoofax name binding language. In *Companion to the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2012, Tucson, AR, USA, October 19 - 26, 2012*. ACM, 2012. doi:10.1145/2384716.2384748.

30 Marieke van der Tuin, A. Bastiaan Reijm, Tim K. de Jong, and Jeff Smits. WebLab project. Bachelor's thesis, Delft University of Technology, July 2013. URL: http://resolver.tudelft.nl/uuid:bb2d7a13-1bef-4545-bca0-f2b084a04240.

31 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, 2016. doi:10.1145/2847538.2847543.

32 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *PACMPL*, 2(OOPSLA), 2018. doi:10.1145/3276484.

33 Elmer van Chastelet. A domain-specific language for internal site search. Master's thesis, Delft University of Technology, 2013.

34 Elmer van Chastelet, Eelco Visser, and Craig Anslow. Conf.Researchr.Org: towards a domain-specific content management system for managing large conference websites. In *OOPSLA*, pages 50–51, 2015. doi:10.1145/2814189.2817270.

35 Sander van der Burg and Eelco Dolstra. Disnix: A toolset for distributed deployment. In *Third International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3)*, September 2010.

36 Tim van der Lippe, Thomas Smith, Daniël A. A. Pelsmaeker, and Eelco Visser. A scalable infrastructure for teaching concepts of programming languages in Scala with WebLab: an experience report. In *SCALA*, pages 65–74, 2016. doi:10.1145/2998392.2998402.

37 Arie van Deursen, T. B. Dinesh, and Emma van der Meulen. The ASF+SDF meta-environment. In *amast*, pages 411–412, 1993.

38 Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for dynamic semantics specification. In *RTA*, pages 365–378, 2015. doi:10.4230/LIPIcs.RTA.2015.365.

39 Sander Vermolen. *Software Language Evolution*. PhD thesis, Delft University of Technology, Delft, The Netherlands, October 2012.

40 Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, August 1997.

41 Eelco Visser. Meta-programming with concrete object syntax. In *GPCE*, pages 299–315, 2002. doi:10.1007/3-540-45821-2_19.

42 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007. doi:10.1007/978-3-540-88643-3_7.

43 Eelco Visser. Performing systematic literature reviews with Researchr: Tool demonstration. Technical Report TUD-SERG-2010-010, Software Engineering Research Group, Delft University of Technology, Delft, The Netherlands, May 2010. URL: http://resolver.tudelft.nl/uuid:22b480a7-d09e-4ae6-abe7-9a5769e03c2b.

44 Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, pages 13–26, 1998. doi:10.1145/289423.289425.

45 Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. Declarative specification of template-based textual editors. In *LDTA*, pages 1–7, 2012. doi:10.1145/2427048.2427056.

Reflections on the Birth of Spoofax

Karl Trygve Kalleberg
KolibriFX AS, Oslo, Norway

Abstract

The Spoofax Language Workbench is one of the many successful research projects to come out of Eelco's tremendously productive career, as evidenced by the Most Influential Paper designation awarded at OOPSLA 2020 to the famous 2010 paper by Lennart and Eelco [6]. In connection with the award, Eelco wrote an excellent retrospective [9].

Here, I will reflect briefly on the birth of the Spoofax, subject to omissions and inaccuracies, as the topic may be of interest to hardcore fans. Eelco was my PhD co-supervisor, and I was fortunate to be part of his group during the inception of the project.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Compilers; Software and its engineering → Interpreters

Keywords and phrases Spoofax, Language Workbench, Program Transformation

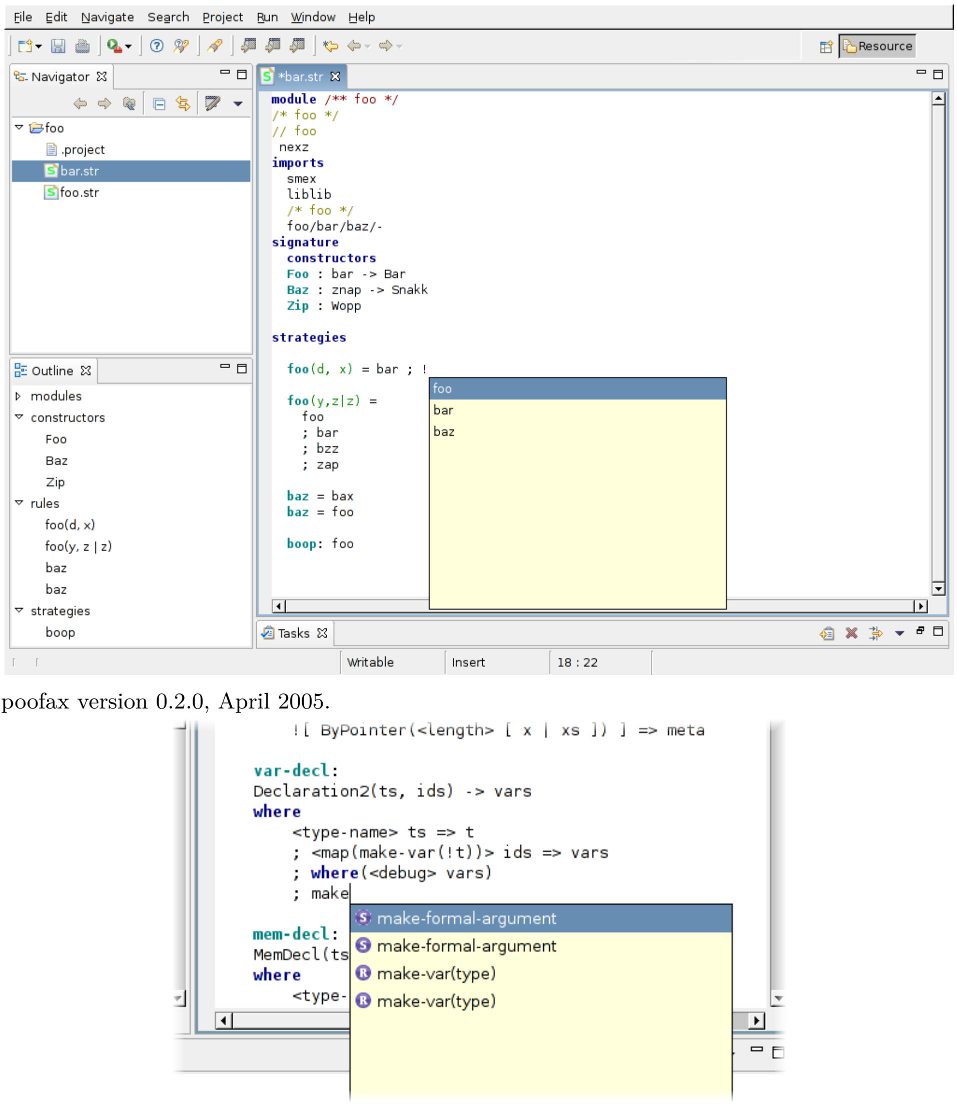
Digital Object Identifier 10.4230/OASIcs.EVCS.2023.14

Acknowledgements I would like to thank Lennart Kats for feedback on the various versions of this manuscript.

A little-known fact is that Spoofax did not actually start out as a language workbench. Its humble beginnings were as an interactive development environment (IDE) for Stratego/XT [1]. During my time with Eelco in Utrecht, he and I had numerous walks in the forest around De Uithof, plotting the total world domination of program transformation in general and Stratego/XT in particular. Whenever interactive transformations came up (refactoring, code completion, semantics-aware navigation, live compilation), I always remarked that we could not really support these use cases with the batch-based Stratego/XT system we had at the time. As some of you know, Eelco was a lover of music, but not into broken records, so I was politely encouraged to put up or shut up.

At the time, building an entire platform for interactive program transformation seemed daunting to me, so I settled on the easier goal of making an interactive editing environment for Stratego based on Eclipse. As with any new project, choosing the name was the hardest part, but once I came up with a pronounceable word that was also free domain name, I registered `spoofax.{com,net,org}` in February of 2005, and then I was off to the races. The first version (0.1.0) of Spoofax was released on March 12 2005, and featured rudimentary syntax highlighting. Over the course of the next few months, I added features such as simple code completion of rule and strategy names, outline, and parenthesis matching (Figures 1a and 1b).

Looking for peer review, Eelco suggested we drive over to Günter Kniesel in Bonn and present what we had at a workshop there, so we did. That was the first time we showed Spoofax to the world. At the workshop, the concept of a language workbench came up again – only we did not use that term yet – but we were very far from realizing anything like that at the time. After summer, Eelco and I had a long discussion on where to go from here. We decided it made sense to reimplement the necessary parts of Stratego/XT in Java, to enable tighter, in-process, integration of the Stratego/XT universe with the Eclipse IDE ecosystem.



```
module /** foo */
/* foo */
// foo
 nexz
imports
  smex
  liblib
  /* foo */
  foo/bar/baz/-
signature
  constructors
  Foo : bar -> Bar
  Baz : znap -> Snakk
  Zip : Wopp

strategies

  foo(d, x) = bar ; !

  foo(y,z|z) =
    foo
    ; bar
    ; bzz
    ; zap

  baz = bax
  baz = foo

  boop: foo
```

(a) Spoofax version 0.2.0, April 2005.

```
     ![ ByPointer(<length> [ x | xs ]) ] => meta

var-decl:
Declaration2(ts, ids) -> vars
where
     <type-name> ts => t
     ; <map(make-var(!t))> ids => vars
     ; where(<debug> vars)
     ; make

mem-decl:
MemDecl(ts
where
     <type-
```

(b) Spoofax version 0.3.0, May 2005.

Figure 1 Screenshots from the early days of Spoofax.

I first put together a Stratego interpreter and then an implementation of SGLR in Java in the autumn of 2005. I plugged both into the rudimentary Spoofax implementation I had already built. This was the first time we had something that could conceivably become the basis of a language workbench, albeit in the crudest form possible. There were a number of really bad kinks. The source code had to be syntactically correct for the parser to work, because, as we all know, parsing stinks. Extending the platform with new transformation logic required manually executing the Stratego compiler on the command line with some unusual switches, then placing the output files in a particular directory monitored by the IDE. An unwieldy tool, with a lot of sharp edges to cut yourself. But it was a start.

My original motivation was for Spoofax to be a modern IDE for Stratego/XT, one that you could extend with your own transformations written in Stratego. In the beginning, Eelco was not convinced that this was a good idea. I cannot know for sure exactly why that was the case, but I remember him as very happy with Emacs and I know that he had limited experience with modern IDEs at the time. Moreover, his previous involvement with the ASF+SDF meta-environment [8] had taught him how difficult and expensive it is to build and maintain desktop applications with graphical user-interfaces. Over time, I believe he saw that, with Eclipse, the UI part came "mostly for free". After we (Lennart, mostly) put together the `sdf2imp` [5] prototype which made it possible to plug any language with an SDF grammar into Eclipse, I believe Eelco realized that he could focus on all of the fun parts from the ASF+SDF meta-environment, with hardly any of the dreaded UI maintenance. This enabled him to formulate a much bigger vision for what this could become, and he leaned on that vision as he guided the project forward after my departure from the group.

Across the gulf of time, minds immeasurably superior to mine regarded the Spoofax prototype with curious eyes, and slowly and surely, they drew their plans to improve it [10]. To name some and omit many (sorry): Lennart single-handedly evolved the early proof-of-concept into a real platform – what would become an actual language workbench [4]; Maartje extended the SGLR implementation and with support for parsing incomplete syntax [2], finally making it practical inside an editor; Gabriel took the concept even further by developing language-parametric methods for deriving interactive programming systems automatically from high-level specifications [7].

It is a testament to his foresight and perseverance that Eelco was able to steadily grow the project from a somewhat programmable Stratego editor into the extensive language ecosystem that Spoofax is today. His knack for attracting talented and hard-working students, and letting them loose on some of the gnarliest problems in the field has been a good recipe for progress. One of the traits I most appreciated in Eelco was his keen balance between the mindset of an engineer and an academic. Constantly the applied computer scientist: pragmatic, practical and visionary. I always found it inspiring to be around him.

After my PhD [3], life took me on a different path, but Eelco and I kept in touch from time to time. I was last in touch with him in late 2021, when we agreed that I would transfer the ownership of the original Spoofax internet domains to his custody. This had been on the docket ever since I left in 2007, but we never got around to it before.

Going forward, it is my hope that the community around Spoofax will continue to thrive in the years ahead, like an epic sax tune that goes on forever. ♪*Never gonna give you up* ♪.

So long, and thanks for all the transformations. I will miss you, my friend.

References

1 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. doi:10.1016/j.scico.2007.11.003.

2 Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems*, 34(4):15, 2012. doi:10.1145/2400676.2400678.

3 Karl Trygve Kalleberg. *Abstractions for Language-Independent Program Transformations*. PhD thesis, University of Bergen, 2007.

4 Lennart C. L. Kats. *Building Blocks for Language Workbenches*. PhD thesis, Delft University of Technology, 2011.

5 Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Generating editors for embedded languages. integrating sglr into imp. In Jurgen J. Vinju and Adrian Johnstone, editors, *Proceedings of the Eight Workshop on Language Descriptions, Tools, and Applications*, volume 238(5) of *Electronic Notes in Theoretical Computer Science*. Elsevier, April 2008.

6 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 237–238. ACM, 2010. `doi:10.1145/1869542.1869592`.

7 Gabriël Konat. *Language-Parametric Methods for Developing Interactive Programming Systems.* PhD thesis, Delft University, 2019.

8 Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2001. `doi:10.1016/S1571-0661(04)80917-4`.

9 Eelco Visser. A Brief History of the Spoofax Language Workbench. `https://eelcovisser.org/blog/2021/02/08/spoofax-mip/`, 2021. [Online; accessed 30-January-2023].

10 Jeff Wayne. The Eve of the War (track 1), The Musical Version of The War of the Worlds. CBS, 1978. Produced by Jeff Wayne.

Eelco Visser as a Typographic Designer

Paul Klint ✉ 🏠
Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

Abstract

While supervising both Eelco's master's thesis and his dissertation I have witnessed an evolution in his taste and style in many aspects of design, ranging from visual and typographical design to software design and even presentation and article design. I give some personal reflections on this evolution.

2012 ACM Subject Classification Software and its engineering → Documentation

Keywords and phrases Typesetting formal specifications, Typographic design, personal recollection

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.15

1 Prologue

The terminology to describe the relationship between a supervisor and a student is troublesome. The phrase "my student" always reminds me of modern slavery, which it probably is, but should one put that in writing or talk about it loudly? The word "supervisor" suggests an asymmetric relationship where the supervisor is all mighty and all knowledgeable and this is also frequently besides the truth.

Having set this straight, I have had the privilege to "supervise" Eelco on various occasions. The first was a side project about the semantics of Eiffel while he was still a computer science student at the University of Amsterdam; it was completed by the end of 1992 [6]. It was a substantial (125 page) document that could have easily served as a Master's Thesis, but not so in Eelco's opinion. Important stylistic properties: a text-only, red front page and inside ASF+SDF specifications printed in a mono-spaced font.

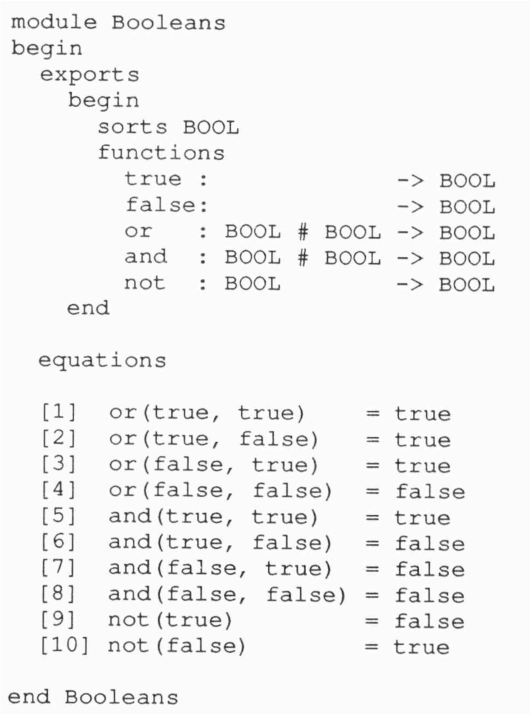
2 Thesis: Thou Shalt Typeset Specifications

Eelco and I had many interests in common, among them context-free parsing, typesetting, and literate programming. My own dissertation (1982, published in 1985 as [3]) was probably the first computer typeset dissertation in The Netherlands and when I a gave a copy to Edsger Dijkstra he said: "This is the first document where I cannot see that it has been produced by a computer". From his mouth, I took that as a compliment.

This is not the place for a complete genealogy of all specification formalisms we have collectively worked on, but some context is useful. It all started with ASF [1], an algebraic specification formalism with an implementation based on term rewriting. Next came ASF+SDF [5] that added general context-free grammars and user-defined notation. Its implementation was based on GLR parsing and term rewriting. Eelco used his experience with ASF+SDF and the idea of strategic rewriting [4] to design Stratego [9]. Eelco introduced the idea of scannerless parsing in SDF and later evolved it to SDF3 [2].

In the spirit of literate programming I had been working on a program called ToLaTex that took ASF+SDF specifications and turned them into LaTeX source. I was particularly proud about the use of the equation and table features of LaTeX to typeset our specifications. Figure 1 shows a manually formatted ASF specification of the Boolean datatype using a fixed width font. Figure 2 shows the Booleans in ASF+SDF, typeset using ToLaTex.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 15; pp. 15:1–15:5
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



```
module Booleans
begin
  exports
    begin
      sorts BOOL
      functions
        true :                 -> BOOL
        false:                 -> BOOL
        or   : BOOL # BOOL -> BOOL
        and  : BOOL # BOOL -> BOOL
        not  : BOOL            -> BOOL
    end

  equations

  [1]  or(true, true)     = true
  [2]  or(true, false)    = true
  [3]  or(false, true)    = true
  [4]  or(false, false)   = false
  [5]  and(true, true)    = true
  [6]  and(true, false)   = false
  [7]  and(false, true)   = false
  [8]  and(false, false)  = false
  [9]  not(true)          = false
  [10] not(false)         = true

end Booleans
```

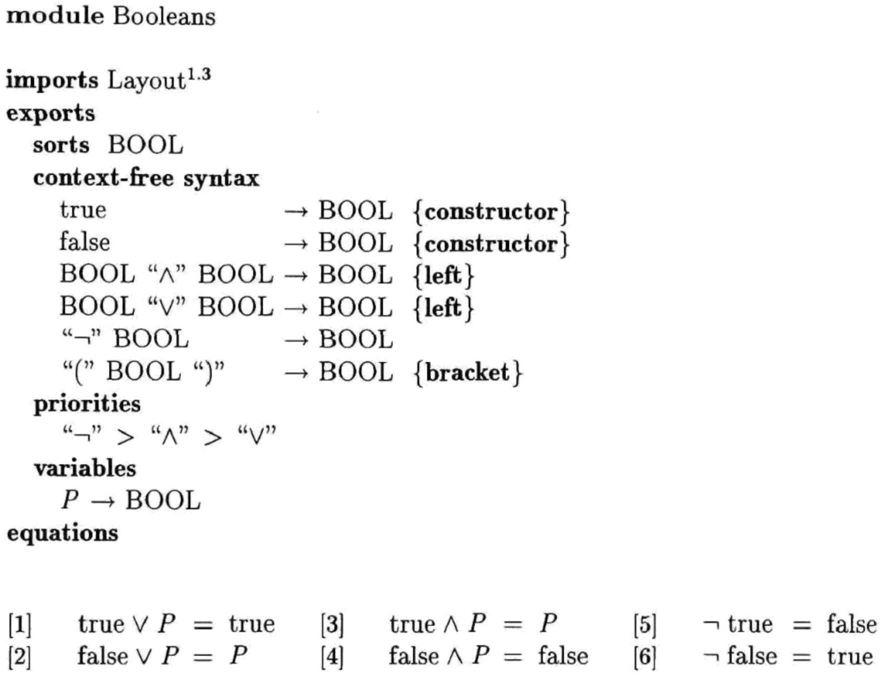
Figure 1 Booleans in ASF, manual layout.

While Eelco started working on what would become his actual Master's Thesis, I was further improving ToLaTex. Eelco got interested to use and enhance it for his thesis. Eelco's typical first step was to split the single Lisp file into 50 small ones. I see many people performing such "modularization" and I never understand them: it is certainly more modular but at the same time overview and convenient editing are gone. Although not amused by Eelco's action, I took it from the bright side: from then on Eelco was the maintainer of ToLaTeX. In 1993 Eelco's Master's Thesis appeared [7]. Another 130 page tome dense with equations, but stylistically different from the Eiffel document: a nice, black-and-white cover that showed his former training as an artist (shown in Figure 4a). Inside, all specifications had been typeset using ToLaTex that by this time supported subscripts, superscripts and the complete zoo of LaTeX escapes for special characters and mathematical symbols. In later years Eelco continued to work on various techniques for prettyprinting and source code formatting [10]. Eelco continued the same style in his dissertation [8]: a green cover with a design that can easily be taken for a bull's eye merged with trees (shown in Figure 4b). And indeed Eelco was aiming at trees. The 383 page tome is, amongst others, a major ToLaTeX showcase: every feature has been used to the max.

3 Antithesis: Thou Shalt Not Typeset Specifications

Despite the relative success of his dissertation Eelco was disappointed. He ventilated his frustration in a post on a programming-related mailing list[1]. A copy of that post has been hanging in the hallway of our institute for years. Eelco was a programmer and his complaint

[1] I recall this from memory. Although I tried, I was not able to retrieve it.



```
module Booleans

imports Layout1.3
exports
  sorts  BOOL
  context-free syntax
    true                  → BOOL  {constructor}
    false                 → BOOL  {constructor}
    BOOL "∧" BOOL         → BOOL  {left}
    BOOL "∨" BOOL         → BOOL  {left}
    "¬" BOOL              → BOOL
    "(" BOOL ")"          → BOOL  {bracket}
  priorities
    "¬" > "∧" > "∨"
  variables
    P → BOOL
equations

[1]   true ∨ P  = true     [3]   true ∧ P  = P       [5]   ¬ true  = false
[2]   false ∨ P = P        [4]   false ∧ P = false   [6]   ¬ false = true
```

Figure 2 Booleans in ASF+SDF, typeset with ToLaTex.

was that due to the fancy typesetting everybody thought that what he was writing about is math and not programming. It is undeniably true that Greek letters, mathematical symbols, and subscripts and superscripts add to the mathematical flavour of a document. Let's also discuss the issue of readability using the example shown in Figure 3 taken from [8]. Although this is a relatively readable module one thing stands out. Due to the incredible syntactic flexibility of ASF+SDF nearly everything is user-definable, this includes names and variables. For instance $\overline{Al}$, al_1^* and al_2^* are just names (declared in one of the imported modules) while the notation suggests otherwise. The nice typesetting reinforces this effect. In hindsight I think that Eelco's frustration resulted from a combination of ASF+SDF's syntactic flexibility, his actual use of that flexibility, and the fancy typesetting offered by ToLaTex. Since then Eelco gradually gave up on typesetting specifications and started to represent them in a typewriter font. Even to this date, his frustration can be read in the writing advice on his website[2]:

> *Don't use a body text font (such as Times or cmr) for code. Preferably use a proportional font (e.g. some typewritter [**sic**] font), which gives neat alignment and indentation. (Using math for typesetting code, as is done in literate Haskell for example, is not for me. I overdosed on that in my PhD thesis.)*

4 Synthesis

Over the last 5-10 years Eelco's focus has gradually shifted from programming to a more formal view on software development, e.g., safety by construction. Unavoidably, with more formality more complex notation enters the scene. In many of his latest papers we see a

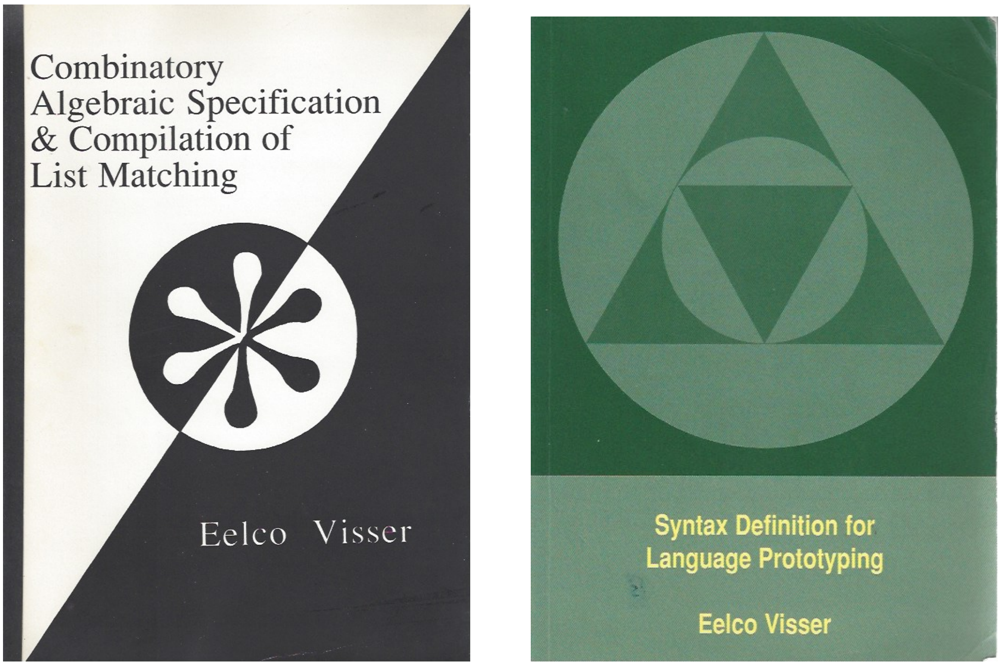
[2] See `https://eelcovisser.org/wiki/teaching/writing/`.

module Alias-Sdf-Projection
imports Alias-Sdf-Syntax$^{9.2.1}$
exports
  context-free syntax
    Aliases "++" Aliases → Aliases   {**right**}
    "Al" (Grammar) → Aliases
    "$\overline{\text{Al}}$" (Grammar) → Grammar
equations

The function 'Al' gives all alias declarations of a grammar, '$\overline{\text{Al}}$' the grammar without alias declarations.

$$\begin{array}{lrcl}
[1] & al_1^* \mathbin{+\!\!+} al_2^* & = & al_1^* \; al_2^* \\
[2] & \text{Al}(\text{aliases } al^*) & = & al^* \\
[3] & \text{Al}(\mathcal{G}_1 \; \mathcal{G}_2) & = & \text{Al}(\mathcal{G}_1) \mathbin{+\!\!+} \text{Al}(\mathcal{G}_2) \\
[4] & \text{Al}(\mathcal{G}) & = & \quad \textbf{otherwise} \\
[5] & \overline{\text{Al}}(\text{aliases } al^*) & = & \emptyset \\
[6] & \overline{\text{Al}}(\mathcal{G}_1 \; \mathcal{G}_2) & = & \overline{\text{Al}}(\mathcal{G}_1) \; \overline{\text{Al}}(\mathcal{G}_2) \\
[7] & \overline{\text{Al}}(\mathcal{G}) & = & \mathcal{G} \quad \textbf{otherwise}
\end{array}$$

Figure 3 A module from Eelco's dissertation.


Combinatory Algebraic Specification & Compilation of List Matching
Eelco Visser
Syntax Definition for Language Prototyping
Eelco Visser

(a) Master's Thesis.

(b) Dissertation.

Figure 4 Covers of Eelco's Master's Thesis and Dissertation.

fusion of programming notation and logic notation. A case in point, is the use of syntax highlighting in logical formulae. I see this as an expression of his desire to still keep a link with the programming domain *per se*. He never returned to the literate programming style enabled by ToLaTeX.

5 Epilogue

In the Netherlands we used to have the tradition (these days abandoned at some universities) that a dissertation should be accompanied by a number of “propositions” that summarized the results of the dissertation. The last proposition was supposed to be less serious. I cannot think of a better ending than Eelco’s last proposition (translated from Dutch):

> *The use of “hi” when saying goodbye comes from the realization that every ending is also a beginning.*

References

1 J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

2 Luis Eduardo de Souza Amorim and Eelco Visser. Multi-purpose syntax definition with SDF3. In Frank S. de Boer and Antonio Cerone, editors, *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*, volume 12310 of *LNCS*, pages 1–23. Springer, 2020.

3 P. Klint. *A Study in String Processing Languages*, volume 205 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985. doi:10.1007/3-540-16041-8.

4 Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing. Springer-Verlag, September 1997.

5 Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1997.

6 Eelco Visser. Syntax and static semantics of Eiffel. a case study in algebraic specification techniques, December 1992. URL: https://eelcovisser.org/publications/1992/Visser92.pdf.

7 Eelco Visser. Combinatory algebraic specification & compilation of list matching. Master’s thesis, Department of Computer Science, University of Amsterdam, June 1993. URL: https://eelcovisser.org/publications/1993/Visser93.pdf.

8 Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997. URL: https://eelcovisser.org/publications/1997/Visser97.pdf.

9 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.

10 Eelco Visser and Mark G. J. van den Brand. From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, July 1994. URL: http://ivi.fnwi.uva.nl/tcs/pub/reports/1994/P9420.ps.Z.

Typed Multi-Language Strategy Combinators

James Koppel
MIT, Cambridge, MA, US

Abstract

Strategy combinators (also called *strategic programming*) are a technique for modular program transformation construction invented by Bas Luttik and Eelco Visser, best known for their instantiation in the Stratego language. Traditional implementations are dynamically typed, and struggle to represent transformations that can be usefully applied to some types, but not all.

We present the design of our strategy-combinator library COMPSTRAT, a library for type-safe strategy combinators which run on Patrick Bahr's *compositional datatypes*. We show how strategy combinators and compositional datatypes fuse elegantly, allowing the creation of type-preserving program transformations which operate only on datatypes satisfying certain properties. With this technique, it becomes possible to compactly define program transformations that operate on multiple programming languages. COMPSTRAT is part of the Cubix framework and has been used to build four program transformations, each of which operates on at least three languages.

2012 ACM Subject Classification Software and its engineering → Translator writing systems and compiler generators; Software and its engineering → General programming languages

Keywords and phrases program transformation, strategic programming

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.16

Supplementary Material *Software (Source Code)*: `https://github.com/cubix-framework/cubix/tree/master/compstrat`; archived at `swh:1:dir:710191ca2e67e6f922bccc5a9bbc1088699d618f`

Funding This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374.

1 Introduction

If a program transformation can be written, it can be written using rewrite rules. Rewrite rules become usable when paired with strategies, functions that apply them. Building strategies is made economical by *strategy combinators* [8, 16, 15].

Typed functional programming has shown many advantages for building language tools. Unsurprisingly, there have been many attempts to bring strategy combinators to typed functional languages, including STRAFUNSKI [7], KURE [2, 11], and Scrap Your Boilerplate (SYB) [5] and its extension RECLIB [10]. All of these grapple with the same challenge: how to build transformations that can run on many types, while preserving the guardrails of strong static typing. All known solutions to this problem use some form of dynamic typing. And in that, we argue, all of them go too far.

Figure 1 illustrates the problem of functions which are "too" dynamically-typed. Each framework defines its own type for generic rewrites, which we call Rewrite x; Table 1 gives the encoding for each framework. The first three – STRAFUNSKI, SYB, and RECLIB – are maximally dynamically typed, in that the type system system allows Rewrite x to be applied to nearly any type, requiring runtime type-casing to constrain it. Each of these are dynamically typed. Now consider the code in Figure 1, which can be thought of as a fragment of an application that invokes some refactoring transformation, storing the new program and a message in a Result data structure. Although the implementation of doRefactoring may look innocuous to someone encountering it without exact memory of the Request type, it contains a deadly bug. And yet this code typechecks verbatim in SYB! STRAFUNSKI and RECLIB

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 16; pp. 16:1–16:9
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Previous Haskell libraries for strategy combinators.

Library	Year	Encoding of `Rewrite x`
STRAFUNSKI	2003	(**Monad** m, Data x) ⇒x →m x
SYB	2003	(**Monad** m, Data x) ⇒x →m x
RECLIB	2006	(Data x) ⇒ a → x → **Maybe** (a, x) (and many variations)
KURE	2007	(**Monad** m) ⇒c →G →m G where G = GExp Exp | GDecl Decl |...
COMPSTRAT	2013[3]	(**Monad** m, HTraversable f) ⇒Term f l → m (Term f l)

```
myRefactoring :: Rewrite JavaProgram
 ...

data RefactorRequest = Request String JavaProgram
data RefactorResult = Result JavaProgram String

doRefactoring :: RefactorRequest →RefactorResult
doRefactoring (Request p s) = Result p (myRefactoring s) −− Oops
```

Figure 1 Example pitfall of dynamically-typed strategy combinators.

would give an error, but only to point out that the rewrite runs in the **Maybe** monad. After fixing the type errors, the true problem remains: myRefactoring is being run on a string rather than on a program, with no effect. When the transformation being applied is one step in a larger sequence, such a bug can linger undetected. KURE[1] is less dynamically typed, permitting a Rewrite x to only be applied to terms of a custom program type. But in exchange it allows a more elementary class of dynamic typing errors: if a KURE rewrite attempts to replace all identifiers in a program with statements, the malformed output will be detected only at runtime.[2] These contrast the COMPSTRAT representation, which, while also internally allowing dynamic type dispatch, is able to place arbitrary constraints on both the set of nodes and set of sorts supported by a rewrite.

In previous work [4], we created CUBIX[4], the "One Tool, Many Languages" framework which makes it possible to build source-to-source program transformations where the entire target programming language is a type parameter. For example, Figure 2 shows a slightly-simplified[5] version of the top-level code for the test-coverage instrumentation transformation, which inserts extra output for test coverage. Despite supporting five languages, its implementation totals only 202 lines.

In Figure 2, the careful reader familiar with strategy combinators may recognize the allbuR (all subterms, bottom-up Rewrite) combinator from STRATEGO's allbu combinator. Indeed, all serious CUBIX transformations are built using a custom strategy combinator library, COMPSTRAT. And it turns out that the special representation of programs which lies at the core of CUBIX incremental parametric syntax, realized using Patrick Bahr's compositional data types [1] synergizes with strategy combinators in a way that allows both more type

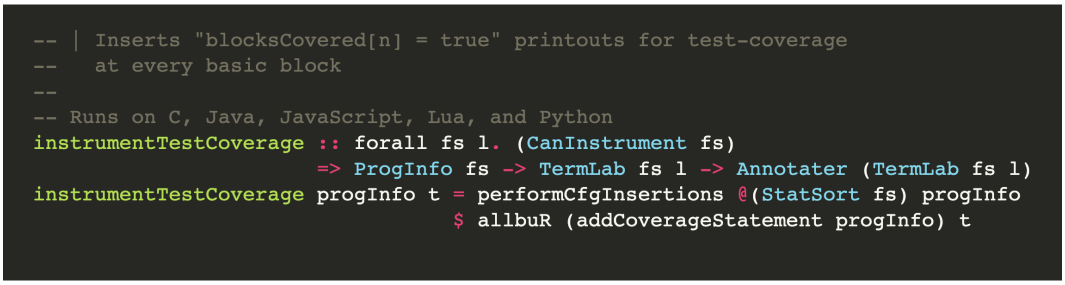
[1] Older solutions in monotyped or untyped languages, namely Stratego itself and JJForester [14], are similar to KURE in that they only express rewrites over arbitrary terms.

[2] In fairness, the recommended APIs of KURE do not permit constructing such a malformed rewrite, although the types permit it.

[3] COMPSTRAT was first implemented in 2013, released on Hackage in 2015, mentioned in the 2018 CUBIX paper, and never explained in an academic paper until now

[4] `http://cubix-framework.com/`

[5] The original contains extra code related to management of label indices.



```
-- | Inserts "blocksCovered[n] = true" printouts for test-coverage
--   at every basic block
--
-- Runs on C, Java, JavaScript, Lua, and Python
instrumentTestCoverage :: forall fs l. (CanInstrument fs)
                       => ProgInfo fs -> TermLab fs l -> Annotater (TermLab fs l)
instrumentTestCoverage progInfo t = performCfgInsertions @(StatSort fs) progInfo
                                  $ allbuR (addCoverageStatement progInfo) t
```

Figure 2 Slightly-simplified top-level code of test-coverage transformation, screenshotted from `www.cubix-framework.com`.

safety and an easier implementation compared to all previous attempts. Using COMPSTRAT, it is natural to define a rewrite where e.g.: the compiler guarantees that expressions will only be transformed into other expressions, and guarantees that the transformation may be run on Python and Java programs but not C.

Previous Cubix papers [9, 4, 3] shied away from discussion of strategy combinators. This paper is the first presentation of the COMPSTRAT library. We shall not present most of its elements – they are built from the basic combinators the same way as in every other strategy combinator library. We shall dwell briefly on its multi-language ability – much of that comes from the representation, and would be the same for any other paradigm built atop Cubix. But we shall present the key ideas of what makes COMPSTRAT different.

2 Background

In this section, we give an abbreviated summary of incremental parametric syntax and compositional data types, the program representation underlying Cubix. The rest of this paper will assume familiarity with strategy combinators.

Incremental parametric syntax is a technique for modularly constructing the datatypes of terms in different languages, in a way where an off-the-shelf representation for a single language can be incrementally refined into one built from modular components, gradually reducing the amount of language-specific code that needs to be written to build a tool for multiple languages. Though it can be presented abstractly, in terms of operators for combining and modifying the signatures of languages, its only known instantiation, in Cubix, is built as an extension of compositional data types [1], which are in turn an extension of data types à la carte [12]. We now present extremely compressed explanations of each of these.

First, the idea of data types à la carte is to make a datatype modular by removing explicit recursion. Instead of defining terms as a recursive datatype such as **data** Exp = Add Exp Exp | Val **Int**, the datatype is defined in *unfixed* form, with the recursive occurrences of Exp replaced by a parameter to be filled in later, i.e.: **data** Exp e = Add e e | Val **Int**. With the recursion removed, the cases of this datatype can be decomposed into fragments which can be recombined in exponentially many variations. Likewise, operations on nodes are defined on individual fragments, and combined into operations on any datatype built out of these fragments. See Figure 3 for a full example.

The encoding of datatypes à la carte produces only unsorted terms. Compositional data types extend this further by allowing multi-sorted terms. The encoding is modified so that everything takes an extra parameter indicating the sort; datatypes are now defined as

```
data Add e = Add e e
data Val e = Val Int
data (f :+: g) e = Inl (f e) | Inr (g e)
data Term f = Term (f (Term f))

type ExpSig = Add :+: Val
type Exp = Term ExpSig

addExample :: Exp
addExample = Term (Inl (Add (Term (Inr (Val 118))) (Term (Inr (Val 1219)))))
```

Figure 3 Using data types à la carte to present the expression 118+1219, with addition and constant nodes defined in separate fragments.

```
data ArithL; data AtomL; data LitL

data Arith t l where
  Add :: t AtomL → t AtomL
                 → Arith t ArithL
data Atom t l where
  Var    :: String   → Atom t AtomL
  Const  :: t LitL   → Atom t AtomL
  Parens :: t ArithL → Atom t AtomL

data Lit (t :: ∗ → ∗) l where
  Lit :: Int → Lit t LitL
```

Figure 4 Example language fragments.

GADTs so that each constructor may only have one sort. Terms in compositional data types have types which look like Term LangSig ExpL, which is read "terms of sort *Exp* in language Lang." See Figures 4 and 5 for a full example.

Incremental parametric syntax is enabled by taking compositional data types and adding a small idea with a huge impact. The key new ingredient in incremental parametric syntax is to control the subsorting relationship by including *sort injection nodes*. These nodes allow the tree to transition between language-specific and generic nodes in a controlled fashion, so that terms are now represented as a combination of language-specific and generic nodes. This both allows incremental development to support a new language, deferring the upfront cost to replace all nodes in a language with generic nodes, and also allows for language-specific customization of generic nodes, by e.g.: making it possible to independently specific which nodes in a specific language may be used as the LHS of a generic assignment node. In total, this approach makes it feasible to scale the datatypes à la carte approach to multiple real languages. Indeed, Cubix is the first known framework[6] to do so, with support for C, Java, JavaScript, Lua, and Python [4]. Figure 6 gives an example of such a tree, and Table 2 gives examples of ways to refer to different classes of terms.

[6] GitHub's semantic framework is the second, although they later abandoned this approach, citing difficulties stemming from their monosorted approach [13]

```
data (:+:) f g t l = Inl (f t l) | Inr (g t l)
data Term f l = Term (f (Term f)) l
type LangSig = Arith :+: Atom :+: Lit
type LangTerm = Term LangSig
```

Figure 5 Combining the fragments of Figure 4.

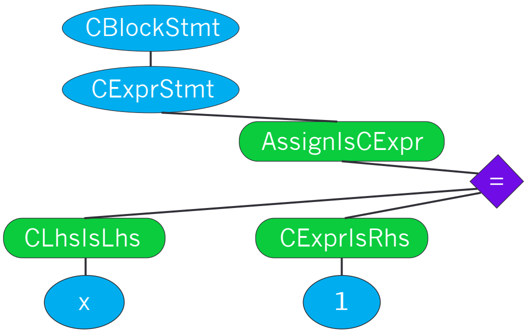


Figure 6 A term in the incremental parametric syntax for C. The ellipses (light blue) represent language-specific nodes; rhombi (purple) represent generic nodes; rounded rectangles (green) represent sort injection nodes.

Table 2 Various term types in Cubix.

Type signature	Description
Term f AssignL	Assignments in any language
Term MJavaSig l	Java terms of any sort
(Assign :<: f) ⇒ Term f IdentL	An identifier in any language that contains generic assignments
Term f (StatSort f)	A statement in any language. The statement sort is language-specific.
(InjF f IdentL PositionalArgExpL , CallAnalysis f) ⇒ Term f IdentL	An identifier in any language which supports a call analysis, and where identifiers may be used as ordinary arguments to functions

One result of this approach is that a single sort may have significance across multiple languages. This means that testing for the sort of a node can be quite useful in strategy combinators, a fact which compstrat exploits.

3 compstrat: Strategy Combinators on Compositional Datatypes

Strategy combinators are an expressive way to build complicated traversal patterns from small building blocks, exemplified by the famous definition of a bottom-up traversal combinator, allbu (t) = **all**(allbu (t)); t. Built on these primitives, implementations of strategy combinator libraries tend to be short, elegant – and identical. Indeed, compstrat provides many identically named combinators to KURE. We thus focus our presentation of compstrat on these building blocks.

Lämmel, Visser, and Visser[6] name a handful of primitives that any strategy combinator library must support. Their list consists of: basic identity and failure strategies, sequential composition, left-biased choice, type-based dispatch, and one-layer traversals applying a strategy to either all or any child of the present node. We shall first define the type of rewrites, from which the basic rewrites and sequencing combinators follow trivially, including the sequencing of rewrites with failure (which is achieved by use of the **Maybe** monad, same as in other Haskell strategy combinator libraries). From there, we turn to discussion of the two other main primitives: one-layer traversal (**all** in Stratego), and type-specific rewriting.

We do simplify some aspects not relevant to showing the unique aspects of compstrat's approach. We present definitions on monads rather than applicatives, and ignore the support for terms with holes. We also present only type-preserving rewrites, ignoring type-altering transformations such as the crush operator.

```
type RewriteM m f l = f l → m (f l)
type Rewrite f l = RewriteM Identity f l

type GRewriteM m f = forall l. RewriteM m f l
type GRewrite f = GRewriteM Identity f
```

Figure 7 Rewrite type.

3.1 The basic encoding

Figure 7 gives the type of rewrites in compstrat.

While straightforward, this is already delivering most of the novel value of compstrat. Consider a rewrite of type Rewrite (Term f) ExpL, which is statically guaranteed to rewrite expressions to expressions in any language. KURE's encoding does not have a type for multi-language rewrites at all. Meanwhile, the encodings of SYB, Strafunski, and RecLib at best permit a rewrite that can run on any type whatsoever. Meanwhile, compstrat's encoding allows adding arbitrary constraints to the languages and sorts supported by a rewriting, e.g.: (CanTransform f) ⇒ Rewrite (Term f) ExpL, allowing high levels of control on where a rewrite may be applied.

Note that the traditional presentation of strategy combinators assumes all rewrites may fail. But here, rewrites can be defined which are statically known not to fail. Only some combinators deal in failable rewrites, indicated by a **MonadPlus** constraint on the m variable.

3.2 One-layer traversal

Applying a rewrite to every child of a node should be simple, and it is: the one-layer traversal primitive is effectively identical to a method of HTraversable from [1], an analogue of the standard Traversable typeclass for higher-kinded terms.

```
allR :: (Monad m, HTraversable f) ⇒GRewriteM m (Term f) →GRewriteM m (Term f)
allR f (Term t) = fmap Term (hmapM f t)
```

Yet this definition, by using a typeclass which generalizes over all tree nodes but not other types, has advantages over the equivalent combinators for the other libraries. The authors of KURE [11] criticized the inflexibility of approaches such as SYB based on Data.Data, where the only way to define a custom traversal pattern is to provide a custom Data instance – but such an instance would violate the laws that Data is supposed to follow![7] Like KURE, compstrat makes it possible to define a custom traversal pattern, as custom HTraversable instances are simpler to write than Data and have fewer constraints. But unlike KURE, users can still obtain the default traversal automatically.

3.3 Specialization and dispatch

Like previous approaches, the use of compositional data types makes it straightforward to define a rewrite which fails except on a single constructor. Unlike previous approaches, compstrat's Rewrite type makes it easy to give a type signature for a rewrite which only runs on a single sort – or a single language, or a family of sorts. For example, here's a rewrite that is statically known to run only on languages which have generic identifiers.

[7] An alternative is to add extra typecases to every rewrite application over every tree that may contain a node where the default traversal is insufficient.

```
-- compstrat
tryR :: (Monad m) ⇒RewriteM (MaybeT m) f l →RewriteM m f l

-- KURE
tryR :: MonadCatch m ⇒Rewrite c m a →Rewrite c m a

-- Closest equivalents in Strafunski
succeed :: Maybe x →x
ifM :: MonadPlus m ⇒m a →(a →m c) → m c → m c
```

Figure 8 Support for failure.

```
vandalize' :: (Ident :<: f) ⇒ Rewrite (Term f) IdentL
vandalize' (project → (Just (Ident s))) = return (iIdent (s ++ "_foo"))
```

It uses the open-sum projection operator, available in all implementations of datatypes à la carte.

COMPSTRAT uses its DynCase class to allow specialization based on sort, as distinguished from the more typical specialized based on specific constructors is based on the DynCase class. It takes a term of an unknown sort b, and possibly returns a proof that b is equal to some known sort a. This makes it possible to lift single-sorted rewrites to multisorted, as in the dynamicR and promoteR combinators later in this section.

```
class DynCase f a where
  -- | Determines whether a node has sort @a@
  dyncase :: f b → Maybe (b :~: a)
```

The DynCase typeclass may look like the built-in Typeable typeclass relied on by SYB, STRAFUNSKI, and RECLIB. But it's different in an important way. A generated instance of this class for the language in Figure 4 looks like this:

```
instance (Arith :<: f) ⇒ DynCase (Term f) ArithL where
  dyncase x = case project x of
                Just (Add _ _) →Just Refl
                _              → Nothing
```

As we can see, dyncase is implementable just as a mundane case match, without terms needing to carry extra runtime-type information, as is required by Typeable. It is used to write isSortR, which fails except at nodes of the desired sort, and dynamicR, which makes a sort-specific rewrite run at all sorts, failing at all save the desired sort.

```
isSortR  :: (DynCase f l, MonadPlus m) ⇒Proxy l →RewriteM m f l'
dynamicR :: (DynCase f l, MonadPlus m) ⇒RewriteM m f l →GRewriteM m f
```

3.4 Flexible Monad Stack

An idea which could be easily added to existing strategy combinator libraries, but strangely isn't, is the flexible monad stack. To explain this, let us look at the type signature of COMPSTRAT's tryR combinator in Figure 8, which takes a failable rewrite and makes it always succeed, and contrast it with its equivalents in other frameworks.

(Note that we found no equivalent in either SYB or RECLIB.)

The COMPSTRAT version takes a rewrite which *may* fail, and returns one which is statically known not to. The other versions input and output rewrites in the same monad. As a result, code running rewrites in the other frameworks must frequently check for failure or call fromJust on code which is known to never fail; clients of COMPSTRAT need not.

tryR is used to define promoteR, which escalates a sort-specific rewrite to run on all sorts, doing nothing at nodes of the wrong sort.

```
promoteR :: (DynCase f l, Monad m) ⇒RewriteM (MaybeT m) f l →GRewriteM m f
promoteR = tryR . dynamicR
```

Other combinators in COMPSTRAT using this idea include allStateR, which accumulates results in a local state monad, and prunetdR, which runs a failable transformation on all nodes except the descendants of transformed nodes.

3.5 Putting it together

We can now show a simple transformation built using COMPSTRAT: using only two lines of code and two lines of type signatures, the vandalize transformation modifies all identifiers in a term of any language that uses CUBIX's generic identifier node. Note that it (1) is guaranteed to never fail (2) can only be run on terms, and only terms in applicable languages, and (3) is guaranteed to never change the sort of what it runs on (e.g.: when run on an expression, returns an expression; when run on a declaration, returns a declaration). It uses vandalize' defined above; here is the new part:

```
vandalize :: (Ident :<: f, HTraversable f) ⇒ GRewrite (Term f)
vandalize = allbuR (promoteR (addFail vandalize'))
```

4 Examples and Applications

The original CUBIX paper presented four program transformations, all of which use COMPSTRAT, although two of them only use it in a small way. We defer to the original paper [4] for a description of these transformations. Here we present code snippets taken from these transformations showing interesting uses of COMPSTRAT.

Specifically, we show pieces of the Hoist transformation, which lifts variable declarations to the top of their scope. It uses the custom combinator transformOuterScope f g, which runs g on all nodes outside a scope-delimiting block, and f on all nodes inside it. New combinators used here include guardBoolT, which takes an operation returning a boolean and converts it to a failable rewrite; guardedT, essentially an if-statement in strategy-combinator form; and addFail :: Rewrite m f l → Rewrite (MaybeT m) f l, which treats a non-failable rewrite as failable. Another thing to note here is the use of the CanHoist constraint, which ensures this combinator (and others in the same file) may only be run on applicable languages.

```
transformOuterScope :: ( MonadHoist f m, CanHoist f)
                    ⇒ GRewriteM m (Term f) →GRewriteM (MaybeT m) (Term f)
                                            → GRewriteM m (Term f)
transformOuterScope f g = tryR (
   guardedT (guardBoolT (isSortT (Proxy :: Proxy BlockL))) (addFail (alltdR f))
                                                            (addFail g)
   >=> allR (transformOuterScope f g))
```

Along with the sort-specific rewrites addIdents, transformBlockItems, and transformStatSorts, this combinator is now used to define the core of the Hoist transformation.

```
hoist = transformOuterScope
          (promoteR addIdents)
          ((dynamicR transformBlockItems)
            >+> (dynamicR transformStatSorts))
          items
```

5 Conclusion

We have presented a library for type-safe strategy combinators. Unlike all previous approaches, COMPSTRAT allows programmers to build transformations that are statically guaranteed not to fail, statically guaranteed to preserve sorts, and restricted at the type level to only run on (certain classes) of terms rather than on arbitrary datatypes. COMPSTRAT is available from `https://github.com/cubix-framework/cubix/tree/master/compstrat`, a part of the CUBIX framework.

References

1 Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 83–94, 2011.

2 Andy Gill. A Haskell Hosted DSL for Writing Transformation Systems. In *Domain-Specific Languages*, pages 285–309. Springer, 2009.

3 James Koppel. One CFG-Generator to Rule Them All. `https://www.jameskoppel.com/files/papers/cubix_cfg.pdf`. Accessed: 2022-10-28.

4 James Koppel, Varot Premtoon, and Armando Solar-Lezama. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):122, 2018.

5 Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

6 Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. `https://www.researchgate.net/publication/277289331_The_Essence_of_Strategic_Programming`, 2002.

7 Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *International Symposium on Practical Aspects of Declarative Languages*, pages 357–375. Springer, 2003.

8 Bas Luttik and Eelco Visser. Specification of Rewriting Strategies. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, pages 1–16, 1997.

9 Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1066–1082, 2020. `doi:10.1145/3385412.3386001`.

10 Deling Ren and Martin Erwig. A Generic Recursion Toolbox for Haskell, Or: Scrap Your Boilerplate Systematically. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 13–24, 2006.

11 Neil Sculthorpe, Nicolas Frisby, and Andy Gill. The Kansas University Rewrite Engine: A Haskell-Embedded Strategic Programming Language with Custom Closed Universes. *Journal of Functional Programming*, 24(4):434–473, 2014.

12 Wouter Swierstra. Data Types à la Carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

13 Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. Fusing Industry and Academia at GitHub (experience report). *arXiv preprint*, 2022. `arXiv:2206.09206`.

14 Arie van Deursen and Joost Visser. Building Program Understanding Tools Using Visitor Combinators. In *Proceedings 10th International Workshop on Program Comprehension*, pages 137–146. IEEE, 2002.

15 Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies: System Description of Stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, pages 357–361. Springer, 2001.

16 Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building Program Optimizers with Rewriting Strategies. *ACM Sigplan Notices*, 34(1):13–26, 1998.

Eelco Visser – An Exceptional SLE Researcher

Ralf Lämmel
Universität Koblenz, Germany

Abstract

These notes honor Eelco Visser as an exceptional member of the *Software Language Engineering* (SLE) community. The notes are authored from the perspective of an SLE co-founder and a continuous SLE supporter. As an inevitable side effect, a short history of SLE is captured. The commemoration begins with Eelco's role in launching the SLE conference. The commemoration continues with Eelco's contributions as an author and his involvement with running the conference and working towards an *SLE Body of Knowledge* (SLEBoK). The commemoration ends with recalling Eelco's role as the de-facto SLE photographer.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Eelco Visser, SLE, SLE conference, Software Language Engineering, SLEBoK

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.17

1 Eelco Visser's role in launching the SLE conference

The *Software Language Engineering* (SLE) conference[1,2] had its first edition in 2008, but it was getting off the ground in 2007 by an announcement at the ATEM 2007 workshop [13][3] at MODELS 2007. Formally, SLE was founded by a joint effort of the ATEM and LDTA communities.[4,5]

The ATEM organizers were very well aware of Eelco Visser's potential role in making the SLE launch successful. This was one of the reasons why he was invited as a panelist; see Fig. 1 for the final email that was sent to the ATEM 2007 panelists.

In 2007, the broader PL/SE/MODELS+ community – to the extent it was even aware of the emerging SLE effort – was not sure about the scope or the unique role of SLE in the landscape of venues and communities. To many, it felt that there were just enough conferences already to cover all SLE topics. Obviously, the SLE founders felt differently.

These birth pains are also somewhat visible from the questions for the panelists, as we tried to gather additional expert support for the SLE cause. Eelco prepared notes for the panel; see Fig. 2. (The notes are formatted and elisions are applied to facilitate the inclusion into the paper at hand.) On the grounds of the figure, the following claims can be supported:

1. Eelco's definition of SLE (Q1) demarcates SLE from POPL and PLDI in that SLE is understood as not focusing on foundations and implementation of programming languages. I assume that the mentioning of "abstractions" hints at DSLs (see also Eelco's answer to

[1] https://www.sleconf.org

[2] https://dblp.org/db/conf/sle/index.html

[3] ATEM is META (the subject of MODELS) backwards (in "reverse"); ATEM started off with reverse engineering focus on metamodels and schemas to arrive (by 2006) at the much broader scope of SLE with coverage of modelware, schemaware, grammarware, and ontoware.

[4] More precisely, based on the composition of the initial SLE steering committee, SLE was founded by a combination of the ATEM representatives (Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter) and LDTA representatives (Mark van den Brand, Görel Hedin, and Eric Van Wyk) as well as James Cordy. ATEM was immediately "subsumed" by SLE in 2008, whereas LDTA's subsumption by SLE was also eventually completed – after a few more LDTA workshop editions.

[5] http://www.sleconf.org/2008/

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 17; pp. 17:1–17:13
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

From: R. Lämmel <...>
Date: Wed, Sep 26, 2007 at 12:52 AM
Subject: Input on the ATEM panel
To: E. Visser <...>, T. Kuehne <...>, A. Kleppe <...>, K. Czarnecki <...>
Cc: A. Winter <...>, D. Gasevic <...>, J.-M. Favre <...>

Dear panelists,

we look forward the panel on "Grand Challenges in Software Language Engineering", which is going to be the closing event at ATEM 2007, which is the 4th International Workshop on (Software) Language Engineering (SLE). The following text is meant to help you preparing for the panel.

[Parts elided – RL]

I will moderate the panel.
We have collected some questions that you may find inspiring.
`https://professor-fish.blogspot.com/2022/11/atem-2007-panel-post-rehosted.html`
[Original link replaced – RL]

Thanks and regards,
Ralf

Figure 1 Invitation of Eelco Visser to the ATEM 2007 panel.

Q6) and the mentioning of "programmer productivity" hints at tools other than common PLDI artifact types. Arguably, Eelco's definition of SLE (Q1) does not directly get into an explanation of the "engineering" bit, but see Eelco's answer to the next question (Q2).

2. Eelco was involved in DSL-related research since his PhD times. Ever since the early work on Stratego [34], Eelco was getting interested in translation, generation, and optimization for DSLs. His belief in the importance of this area shows in the answer to the question what a panelist would regard as a current SLE challenge (Q3). Eelco nominates "safety of code generation" and he points out that GPCE has knowledge in this area, but industry does not use it, which may suggest that "engineering" has to be addressed in this context. The MDA-related responses (Q5 and Q6) also engage with code generation.
3. When being asked about linguistics and usability (Q7), Eelco is more interested in usability than linguistics (but see, e.g., [27, 26, 31, 39, 12, 16, 21] for software linguistics efforts at the SLE venue that occurred afterwards) and he suggests that we should use our own abstractions so that we feel inclined towards investing into usability. That's exactly Eelco Visser's approach; he is in a rather small, top league of building "stacks of DSLs for metaprogramming" to address, eventually, domains such as web programming.
4. When being asked about "language entropy" (as I would like to call it now) and how to manage it (Q8 and Q9), Eelco doesn't engage with the "philosophical" claim of mine that we may just have too many languages for no good reason. More pragmatically, he combines hopes in "standardization", the "market", and – quite insightfully – the ratio of effort to introduce and to maintain a language versus the productivity it enables.
5. When being asked about his "pet kind of language" (Q10), Eelco refers to (languages for) the "*easy* implementation of IDEs" or what is best captured under the umbrella of language workbenches subsequently [9] (with the actual challenge being discussed by the communities as early as 2010); Eelco was very early in calling out this research direction.

Of course, Eelco Visser helped with launching the SLE conference not only by serving on the "formational" ATEM 2007 panel; in the process of setting up the SLE conference, he also advised on organizational aspects and community engineering.

Q1: "My definition of SLE in elevator speech"

Building tools for building abstractions for improving programmer productivity.

Q2: "SLE finally provides a home for xyz"

A systematic mapping of language engineering tradeoffs. [...]

Q3: "An SLE challenge, readily waiting"

Safety of code generation. In the GPCE community, there is a lot of knowledge about safe code generation; syntax safety, type safety. In the industrial code generation community [...], code generation is done using text-based template engines.

Q4: "Any useless kind of language in sight?"

Lots, but I don't tend to remember these.

Q5: "A praise on MDA as an SLE target"

For me, the contribution of MDA is that it has made high-level programming / modeling and code generation back on the agenda.

Q6: "Clearance sale for programming languages"

Forget about MDA. Develop good run-time systems in a high-level PL. Build a stack of DSLs starting with technical domains, gradually covering more and more aspects of application domains.

Q7: "Connections of SLE to linguistics et al."

Linguistics: been there, not done that. Usability: yeah. That's a general problem for language design. Basic approach: eat your own dog food (then you'll work on usability). We/I don't have the tools and time for doing large scale experiments.

Q8: "Software language engineers to care for IT entropy"

Standardization is claimed to be an important tool to avoid entropy, but, realistically, what is the lifetime of programming paradigms/languages. Any program, in any language, will be legacy in a matter of years (say 5), [...].

Q9: "If you were a congressman on planet HOPL ..."

Let's quote (allegedly) Charles Simonyi: "The market will decide".
The big question: where is the tradeoff between effort required to implement a language and the size of the community of users that make this effort pay off. If it is very easy to define a language, a community size of 1 may be feasible (if making the language is cheaper than make the code that is generated from it). [...]

Q10: "Specific engineering challenges for your pet kind of languages"

Easy implementation of IDEs for textual languages.
Independent extensibility of transformations.

Figure 2 Eelco Visser's notes for the ATEM 2007 panel – with some elisions ("[...]").

2 Eelco Visser's contributions to the SLE conference as an author

Here are two basic facts based on DBLP's records for Eelco Visser and SLE 2008–2021:

- Eelco is the top-publishing SLE author: 18 papers.
- SLE is Eelco's top-targeted venue.[6]

Despite these numbers, Eelco Visser should not be claimed by SLE. Eelco is an exceptional member of several other communities, which often also intersect with SLE – in particular GPCE, OOPSLA, IFIP WG 2.11, and IFIP WG 2.16.

[6] DBLP lists 19 OOPSLA vs. 18 SLE papers, but several of the OOPSLA papers are extended abstracts.

Cite	Year	(Simplified) title	Syntax	Semantics	Typing	Transformation	IDE	Theory	Evaluation
[29]	2020	Gradually typing strategies	×	×	⬤	⬤	×	•	●
[6]	2018	Specification of indentation rules	⬤	×	×	×	×	•	⬤
[18]	2018	Migration to incremental computing	×	×	×	⬤	×	×	⬤
[8]	2018	Migration to a language workbench	×	×	×	●	⬤	×	•
[7]	2017	Priority conflicts	⬤	×	×	×	×	×	⬤
[28]	2017	Dataflow analysis specification	×	●	●	×	×	•	●
[5]	2016	Code completion	⬤	×	×	×	●	⬤	•
[17]	2014	Role-based data modeling and navigation	×	•	•	×	×	●	×
[9]	2013	Conclusions on the LWB challenge	•	•	•	•	●	×	⬤
[37]	2013	Incremental name and type analysis	•	●	⬤	×	⬤	×	⬤
[24]	2012	Name binding and scope rules	•	●	⬤	×	•	×	•
[4]	2011	Layout preservation	•	×	×	⬤	•	⬤	⬤
[32]	2011	Reconstruction of metamodel evolution	●	×	×	•	×	●	•
[22]	2010	Disambiguation of meta programs	⬤	×	×	•	•	•	•
[15]	2009	Data validation & UI concerns in WebDSL	•	×	•	•	×	×	×
[3]	2009	Parser error recovery	⬤	×	×	×	•	×	●
[20]	2009	Retargetable DSLs	×	●	×	●	×	×	•
[1]	2008	Parse table composition	⬤	×	×	×	×	•	●

Figure 3 Eelco Visser's SLE papers – with manually assigned codes.

This section though focuses on Eelco's "SLE profile". Fig. 3 lists Eelco's SLE papers and attaches some info based on a simple coding scheme to organize these papers in terms of topic and methodology.[7] In particular, I manually assigned codes for the following SLE-related *topics*:

Syntax Grammars (or metamodels or schemas) and their implementation.

Semantics Dynamic and translational semantics.

Typing Static semantics, type checking, and semantics analysis.

Transformation Program transformation and metaprogramming overall.

IDE IDE support and language workbenches.

Further, I manually assigned codes regarding *research methodology*:

Theory PL-like metatheory or other forms of formal analysis.

Evaluation Benchmarking or empirical study (e.g., a controlled experiment).

[7] This section does not hold up to the standard of a systematic literature survey. The codes were extracted without any robust approach such as the systematic use of search strings. It helped though that the number of papers is manageable and the papers at hand are more or less familiar to me.

For each code, I manually assigned a *relevance* – to express how relevant the topic or the methodology is for the paper at hand:[8]

- × (Nearly) none
- • Observable, but limited relevance
- ● Significant relevance
- ⬤ Very significant relevance

The SLE "medley" of Fig. 3 gives rise to the following observations or claims:

1. Eelco's contributions to SLE are concerned more often with syntax than with semantics, typing, or transformation. This is also due to the fact that some of the more semantical and transformational contributions went to other venues such as GPCE and OOPSLA. However, Eelco's long-established interest in transformation/metaprogramming and his more recent engagement with semantic analysis also shows clearly at SLE.
2. The general trend – in the broader SE/PL communities – to require a proper form of validation shows in the increasing relevance of "Theory" and "Evaluation" in the table. In a few cases, the papers also "outsourced" validation to extra publications. Eelco focused on building tools and proving their utility; "Evaluation" is more common than "Theory".
3. Despite the IDE topic being foreseen in 2007 (see Sec. 1), it took 6 years for the topic to materialize in Eelco's publication record – at least as far as SLE is concerned, but see, of course, publications elsewhere on Spoofax from Eelco's research team such as [23]. This is in part a testament to the speed of development of research topics, but also proof of other venues' grab on thriving topics.
4. Despite "safety of code generation" being called out in 2007 as an SLE topic (see Sec. 1), Eelco preferred other venues for this topic (see, e.g., [33, 10, 11, 19]). When I started the coding scheme for Fig. 3, I expected to encounter the need for additional codes such as a potential code "code generation" as part of both "Semantics" and "Transformation". However, the initial coding scheme was not ever refined because to be me it seemed that no additional (sub-) categories arose in the Eelco's SLE portfolio.

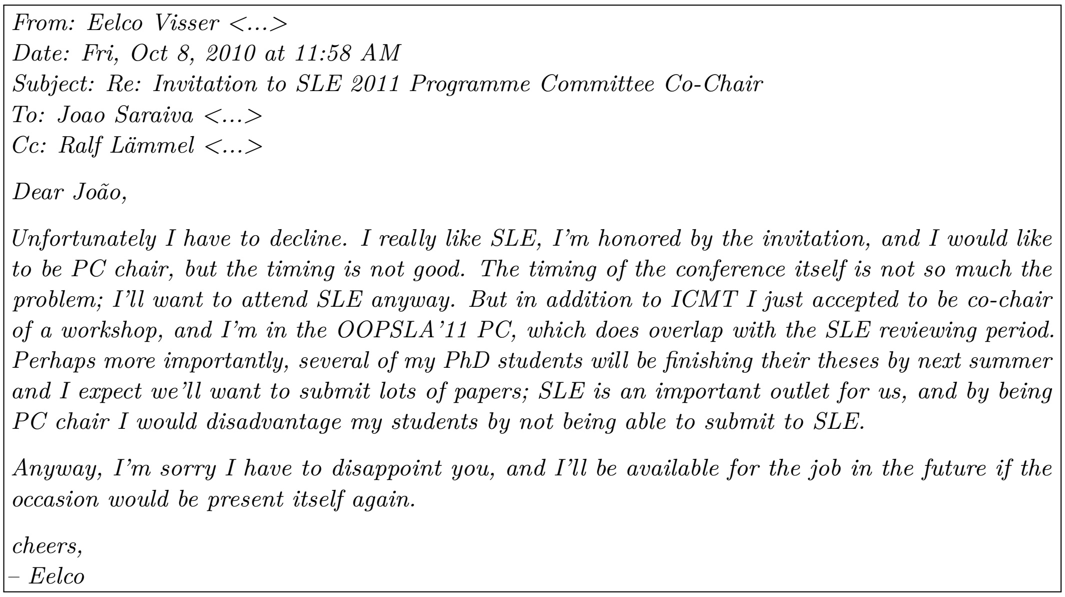
Jean-Marie Favre – possibly the lead founder of SLE, definitely my key mentor regarding technological space travel and integration – told me around 10 years ago, in a discussion of "work-life balance", that we are not the sum and not the count of our papers – certainly not so for our loved ones. Still for the "extended" SLE community, Eelco Visser's publication record and the contributions he (with his many collaborators) made to the field are of a leadership significance – inside and outside the SLE scope.

3 Eelco Visser's involvement in running the SLE conference

Eelco Visser helped with running the SLE conference over the years in several ways:

1. Chair SLE in 2021 [35].
2. Serve on the PC every now and then.
3. Submit to SLE like nobody else (see Sec. 2).
4. Support the co-location between GPCE and SLE.
5. Be available as a peer ever since launch (see Sec. 1).
6. Contribute to conference organization / online existence for SLE.
7. Attend and engage in meetings of the GPCE and SLE steering committees.

[8] Please note that "relevance" of a SLE-related code requires that the paper aims at a contribution to the relevant "research field". For instance, just because a paper includes a "grammar" for a DSL, this does not imply a relevance for the code "Syntax", whereas a fundamental contribution to "parsing" would count as relevant.



From: Eelco Visser <...>
Date: Fri, Oct 8, 2010 at 11:58 AM
Subject: Re: Invitation to SLE 2011 Programme Committee Co-Chair
To: Joao Saraiva <...>
Cc: Ralf Lämmel <...>

Dear João,

Unfortunately I have to decline. I really like SLE, I'm honored by the invitation, and I would like to be PC chair, but the timing is not good. The timing of the conference itself is not so much the problem; I'll want to attend SLE anyway. But in addition to ICMT I just accepted to be co-chair of a workshop, and I'm in the OOPSLA'11 PC, which does overlap with the SLE reviewing period. Perhaps more importantly, several of my PhD students will be finishing their theses by next summer and I expect we'll want to submit lots of papers; SLE is an important outlet for us, and by being PC chair I would disadvantage my students by not being able to submit to SLE.

Anyway, I'm sorry I have to disappoint you, and I'll be available for the job in the future if the occasion would be present itself again.

cheers,
– Eelco

Figure 4 Eelco Visser's NO to the invitation of co-chairing the PC of SLE 2011.

Eelco eventually chaired the SLE conference in 2021, but he was surely asked more than once over the years. Much is known and documented about Eelco's skills to lead and to mentor; I want to include a more concealed entry here while covering an important skill nevertheless. That is, I was able to find one record where he kindly and thoughtfully declined; see Fig. 4. Here is what I call "**the discipline of declining**":

1. Cheer up the inquirers!
2. Don't accept a committee/chairing duty, if your schedule is too crowded already!
3. Make use of self-advertisement to point out what other important duties are in the way.
4. Optionally, tell them you are available in the future. Keep the door open, if you mean it.

4 Eelco Visser's contributions to SLEBoK

SLE, by its very definition [14] (also see the CFPs for SLE 2008–2022), aims at the integration of engineering knowledge regarding software languages across technological spaces, use cases, formalisms, etc.; see again Eelco's replies in Fig. 2 (Q1–Q3). Such knowledge integration was eventually understood as building a "*Body of Knowledge*" (BoK) – a notion used by various communities. Since 2012, we speak of the so-called SLEBoK – the *SLE Body of Knowledge*. Eelco Visser was involved in such activities explicitly aiming at knowledge integration in the SLE context, as highlighted below.

Along the lines of the SLEBoK idea, tutorial-style elements were exercised at SLE conferences or other venues (such as SPLASH) over the years. Notably, SLE 2012 in Dresden featured three mini-tutorials meant to support technological space travel – in the sense of explaining one space for an audience more settled in another space.[9] In particular, Eelco Visser and Guido Wachsmuth contributed "*A Guide to Grammarware*"; see Fig. 5.

[9] https://www.sleconf.org/2012/Minitutorials.html

***Eelco Visser and Guido Wachsmuth:* “A Guide to Grammarware”**

We present a guide to grammarware for researchers with a modelware background, discussing the specification and implementation of various language aspects with grammarware technologies. The guide covers concrete and abstract syntax, name binding, types and constraints, model-to-model transformation, code generation, and execution. For each aspect, we point out possible grammarware approaches, connect and compare them to modelware approaches, survey important grammarware research results, and identify current grammarware research questions. Throughout the guide, we use OCL as the example language. The guide is accompanied by an OCL implementation in the Spoofax language workbench.

***Richard Paige, Dimitrios Kolovos and Fiona Polack:* “Metamodeling for Grammarware Researcher”**

A metamodel is [...].

***Giancarlo Guizzardi and Veruska Zamborlini:* “A Common Foundational Theory for Bridging two levels in Ontology-Driven Conceptual Modeling”**

In recent years, there has been a growing interest in the use Foundational Ontologies, [...]

Figure 5 Minitutorials at SLE 2012 – including one by Eelco Visser.

From: Eelco Visser <...>
Date: Sun, Aug 18, 2013 at 10:12 PM
Subject: Re: Thank you for your SLE 2012 tutorials
To: Ralf Lämmel <...>
Cc: Richard Paige <...>

Hi Ralf,

Your proposal looks fine. I realized only after accepting the invitation that the task was rather impossible. The field is huge and it is hard to provide some coverage. At the same time, the basic material is well known to people working in the field. (At least that is always an assumption I have that may not actually hold.) A comprehensive overview of SLE would be great, but it seems a daunting task for a single author. Perhaps it is time for a handbook!?

– Eelco

On Sun, Aug 18, 2013 at 9:04 PM, Ralf Lämmel <...> wrote:

> *Dear Eelco and Richard,*
>
> *[...] some perspective on the relationship between grammar- and model-based approaches to language design [...] While thinking of it, I ran into your excellent tutorials at SLE 2012 [...]. Below you see the proposal that I have in mind for SPLASH. [...]*
>
> *Thanks and regards,*
> *Ralf*

Figure 6 Eelco Visser suggesting there is a need for an SLE handbook.

I ran into those tutorials from SLE 2012 in Dresden only after the conference, when preparing my half-day SPLASH/SLE 2013 tutorial on “*Language Modeling Principles*”[10] where I also tried to travel the spaces, as opposed to sticking to my comfort space (“grammarware”). My tutorial was the starting point for my work on the “Software Languages Book” [25]. I was encouraged by Eelco’s response regarding a draft proposal for the tutorial; see Fig. 6.

[10] `https://2013.splashcon.org/track/splash-2013-Tutorials`

From: Eelco Visser <...>
Date: Mon, Mar 27, 2017 at 11:41 AM
Subject: Re: Personal Invitation to Dagstuhl Seminar 17342
To: Ralf Laemmel <...>, Eric Van Wyk <...>, Benoit Combemale <...>

Hi Ralf, Eric, Benoit,

Thanks for the invitation! Unfortunately I won't be able to attend the meeting, as I'll be traveling to a WG2.16 meeting that week.

I'm looking forward to the outcome of the meeting.

cheers,
– Eelco

Figure 7 Eelco Visser declining SLEBoK Dagstuhl attendance.

It so happens that SLE 2012 in Dresden was also the starting point for the actual SLEBoK label, subject to the SLEBoK workshop organized by Jean-Marie Favre and Jurgen Vinju.[11] Jean-Marie Favre also contributed some SLEBoK-like elements to the summer school SoTeSoLa (Software Technologies and Software Languages) in Koblenz, just a few months earlier.

In 2017, a representative portion of the SLE community met at Dagstuhl [2][12] to properly get going on working out an actual SLEBoK-style collection of articles (a book for a BoK). At the Dagstuhl seminar, we had working groups on the following topics – with the names edited here for clarity:

- Reuse and modularity in specifications of software languages
- Attribute grammars
- A software language survey
- Parsing
- The SLE curriculum

The overall assumption is that such BoK-style efforts would support the continuous and well compartmentalized exchange between SLE researchers and practitioners (both of the "engineer" type). The SLEBoK Dagstuhl report [2] also features an opinion piece by Benoit Combemale "*On the need for a SLEBoK*". Of course, Eelco Visser was invited to the Dagstuhl seminar, but he had to decline for good reasons; it is on the record that he was interested in the outcome; see Fig. 7.

Progress reports on SLEBoK were delivered at subsequent SLE conference editions; the effort stalled a bit due to the pandemic or otherwise. (Mea culpa!) The most recent conference edition (SLE 2022) encouraged SLEBoK-type submissions in the CFP, but only one such publication [30] (by Friedrich Steimann) was included. This publication provides, in my view, a very good example of a suitable format for SLEBoK papers and it will hopefully inspire others to follow.

In the meantime, textbooks such as [36, 25, 38][13,14,15] are of help, too. Perhaps not surprisingly, one of these books is coauthored by Eelco. Given my intimate knowledge of the other two books, I can say with certainty that they wouldn't exist without Eelco's contribution to the community, to its body of knowledge.

[11] `https://www.sleconf.org/2012/SLEBOK_SLE2012.html`
[12] `https://www.dagstuhl.de/17342`
[13] `https://voelter.de/books.html`
[14] `http://www.softlang.org/book`
[15] `http://www.dsl.design`

5 Eelco Visser as the SLE photographer

This is a role never advertised and never formally claimed, but many of us will remember Eelco going around the conference venue and taking great shots. (He was doing this at all events he attended, not just at SLE!) This is another dimension of Eelco's community engagement which will be dearly missed.

From: Ralf Lämmel <...>
Date: Fri, Apr 14, 2017 at 7:26 PM
Subject: Permission for photo
To: Eelco Visser <...>

Hi Eelco,

may I kindly ask for permission to reuse this photo in my upcoming book?

https://www.flickr.com/photos/eelcovisser/4772847104

Thank you,
Ralf

From: Eelco Visser <...>
Date: Fri, Apr 14, 2017 at 7:38 PM
Subject: Re: Permission for photo
To: Ralf Lämmel <...>

Sure, with full credits and if Jean-Marie agrees as well.
[...]
– Eelco

Figure 8 Eelco Visser giving permission to use his 2012 photo of Jean-Marie Favre.

While completing the "Software Languages Book" [25], I ran into a photo of Jean-Marie Favre taken by Eelco, which I wanted to artistically transform with a deep-learning approach for inclusion in the book; see Fig. 8 for the conversation; see the book for a transformed Jean-Marie Favre based on Eelco's photo in Chapter 1; the artwork is also available online.[16]

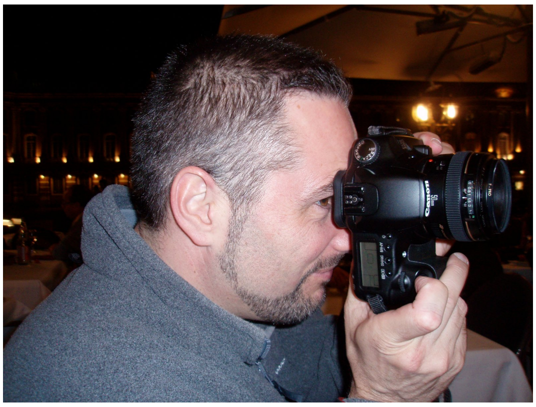


Figure 9 A photo of Eelco in photographer mode at SLE 2008.

[16] http://softlang.uni-koblenz.de/book/artwork.pdf

I want to close these notes in a reasonably "meta" manner – with a (sentence about a) (a reproduction of a) *photo I took of Eelco while he was taking a photo* of something going on at SLE 2008 in Toulouse; see Fig. 9. (In this manner, I also want to allude to the meta-meta focus in Eelco's research: languages of languages.[17])

References

1 Martin Bravenboer and Eelco Visser. Parse table composition. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 74–94. Springer, 2008. `doi:10.1007/978-3-642-00434-6_6`.

2 Benoît Combemale, Ralf Lämmel, and Eric Van Wyk. SLEBOK: the software language engineering body of knowledge (dagstuhl seminar 17342). *Dagstuhl Reports*, 7(8):45–54, 2017. `doi:10.4230/DagRep.7.8.45`.

3 Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, volume 5969 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 2009. `doi:10.1007/978-3-642-12107-4_16`.

4 Maartje de Jonge and Eelco Visser. An algorithm for layout preservation in refactoring transformations. In Anthony M. Sloane and Uwe Aßmann, editors, *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2011. `doi:10.1007/978-3-642-28830-2_3`.

5 Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 163–175. ACM, 2016. URL: `http://dl.acm.org/citation.cfm?id=2997374`.

6 Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages. In David J. Pearce, Tanja Mayerhofer, and Friedrich Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 3–15. ACM, 2018. `doi:10.1145/3276604.3276607`.

7 Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Deep priority conflicts in the wild: a pilot study. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 55–66. ACM, 2017. `doi:10.1145/3136014.3136020`.

8 Jasper Denkers, Louis van Gool, and Eelco Visser. Migrating custom DSL implementations to a language workbench (tool demo). In David J. Pearce, Tanja Mayerhofer, and Friedrich Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 205–209. ACM, 2018. `doi:10.1145/3276604.3276608`.

9 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler,

[17] Thanks to Friedrich Steimann who pointed out some possible transformations of the sentence such that a meta-circular or meta-recursive situation may arise. This requires more reflection on my side.

Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches - conclusions from the language workbench challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. doi:10.1007/978-3-319-02654-1_11.

10 Sebastian Erdweg, Vlad A. Vergu, Mira Mezini, and Eelco Visser. Finding bugs in program generators by dynamic analysis of syntactic language constraints. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 17–20. ACM, 2014. doi:10.1145/2584469.2584474.

11 Sebastian Erdweg, Vlad A. Vergu, Mira Mezini, and Eelco Visser. Modular specification and dynamic enforcement of syntactic language constraints when generating code. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 241–252. ACM, 2014. doi:10.1145/2577080.2577089.

12 Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2010. doi:10.1007/978-3-642-19440-5_21.

13 Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter. 4th international workshop on language engineering (ATEM 2007). In Holger Giese, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2007. doi:10.1007/978-3-540-69073-3_4.

14 Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter. Guest editors' introduction to the special section on software language engineering. *IEEE Trans. Software Eng.*, 35(6):737–741, 2009. doi:10.1109/TSE.2009.78.

15 Danny M. Groenewegen and Eelco Visser. Integration of data validation and user interface concerns in a DSL for web applications. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, volume 5969 of *Lecture Notes in Computer Science*, pages 164–173. Springer, 2009. doi:10.1007/978-3-642-12107-4_13.

16 Jurriaan Hage and Peter van Keeken. Neon: A library for language usage analysis. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 2008. doi:10.1007/978-3-642-00434-6_4.

17 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2014. doi:10.1007/978-3-319-11245-9_14.

18 Daco C. Harkes, Elmer van Chastelet, and Eelco Visser. Migrating business logic to an incremental computing DSL: a case study. In David J. Pearce, Tanja Mayerhofer, and Friedrich Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 83–96. ACM, 2018. doi:10.1145/3276604.3276617.

19 Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Softw. Syst. Model.*, 9(3):375–402, 2010. doi:10.1007/s10270-009-0136-1.

20 Zef Hemel and Eelco Visser. PIL: A platform independent language for retargetable dsls. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, volume 5969 of *Lecture Notes in Computer Science*, pages 224–243. Springer, 2009. doi:10.1007/978-3-642-12107-4_17.

21 Einar W. Høst and Bjarte M. Østvold. The java programmer's phrase book. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 322–341. Springer, 2008. doi:10.1007/978-3-642-00434-6_20.

22 Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Interactive disambiguation of meta programs with concrete object syntax. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 327–336. Springer, 2010. doi:10.1007/978-3-642-19440-5_22.

23 Lennart C. L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 444–463. ACM, 2010. doi:10.1145/1869459.1869497.

24 Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. doi:10.1007/978-3-642-36089-3_18.

25 Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming.* Springer, 2018.

26 Suman Roychoudhury, Sagar Sunkle, Deepali Kholkar, and Vinay Kulkarni. A domain-specific controlled english language for automated regulatory compliance (industrial paper). In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 175–181. ACM, 2017. doi:10.1145/3136014.3136018.

27 Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. Empirical study on the usage of graph query languages in open source java projects. In Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, pages 152–166. ACM, 2019. doi:10.1145/3357766.3359541.

28 Jeff Smits and Eelco Visser. Flowspec: declarative dataflow analysis specification. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 221–231. ACM, 2017. doi:10.1145/3136014.3136029.

29 Jeff Smits and Eelco Visser. Gradually typing strategies. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 1–15. ACM, 2020. doi:10.1145/3426425.3426928.

30 Friedrich Steimann and Marius Freitag. The semantics of plurals. In *Software Language Engineering, International Conference, SLE 2022.* ACM DL, 2022.

31 Eric Umuhoza, Marco Brambilla, Davide Ripamonti, and Jordi Cabot. An empirical study on simplification of business process modeling languages. In Richard F. Paige, Davide Di Ruscio, and Markus Völter, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 13–24. ACM, 2015. doi:10.1145/2814251.2814254.

32 Sander Vermolen, Guido Wachsmuth, and Eelco Visser. Reconstructing complex metamodel evolution. In Anthony M. Sloane and Uwe Aßmann, editors, *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, volume 6940 of *Lecture Notes in Computer Science*, pages 201–221. Springer, 2011. doi:10.1007/978-3-642-28830-2_11.

33 Sander Daniël Vermolen, Guido Wachsmuth, and Eelco Visser. Generating database migrations for evolving web applications. In Ewen Denney and Ulrik Pagh Schultz, editors, *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, pages 83–92. ACM, 2011. doi:10.1145/2047862.2047876.

34 Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, pages 13–26. ACM, 1998. doi:10.1145/289423.289425.

35 Eelco Visser, Dimitris S. Kolovos, and Emma Söderberg, editors. *SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October 17 - 18, 2021.* ACM, 2021. doi:10.1145/3486608.

36 Markus Voelter. *DSL Engineering. Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform, 2013.

37 Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013. doi:10.1007/978-3-319-02654-1_15.

38 Andrzej Wasowski and Thorsten Berger. *Domain-Specific Languages: Effective modeling, automation, and reuse.* Springer, 2023.

39 Vadim Zaytsev and Ralf Lämmel. A unified format for language documents. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 2010. doi:10.1007/978-3-642-19440-5_13.

On the Origins of Coccinelle

Julia Lawall
Inria, Paris, France

Abstract

Coccinelle is a program-transformation system for C code. It has been under development since 2005 and has been extensively used on the Linux kernel. The design of Coccinelle was inspired in part by the author's previous experience in using Stratego/XT, developed by Eelco Visser. This paper reflects on some of Coccinelle's design choices and their relation to Eelco Visser's work.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases Linux kernel, Coccinelle, Stratego/XT, program transformation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.18

Supplementary Material *Software (Source Code)*: `https://github.com/coccinelle/coccinelle` archived at `swh:1:dir:94194d5e17db6802aeb51a826179b95a6f4f1808`

Acknowledgements The author wishes to thank Gilles Muller for his collaboration on the design, application, and maintenance of Coccinelle. The author would also like to thank the anonymous reviewers for their helpful feedback, particularly on historical matters, and Olivier Danvy for help in improving the clarity of the presentation.

1 Introduction

Program transformation has a long history [4, 20, 33]. Anyone who loves to write code also, somewhat ironically, begins to wonder whether a tool could write code for them, or at least perform the repetitive code transformations that are inevitably required as design choices change and a system evolves. This wondering has led generations of researchers to investigate the design of tools to automate various kinds of program transformations. Still, many of these approaches have been designed around toy languages or required users to express transformations at the abstract-syntax tree level, and are thus not practical for use on real, large software projects.

In 2004, the author, with Gilles Muller, began to investigate the problem of how to migrate Linux kernel device drivers from Linux kernel version 2.4 to Linux kernel version 2.6. At that time, the even numbered versions of the Linux kernel were considered to be *stable* releases, and were maintained with bug fixes, in parallel to the odd numbered versions, namely Linux kernel version 2.5, in which new features were integrated. Linux 2.4 was first released in January 2001 and Linux 2.6 in December 2003, amounting to a substantial time lapse, with many intervening changes across the code base. Manually updating code, such as device drivers maintained outside the Linux kernel code base, for use with Linux version 2.6 could be a substantial challenge. Our study of changes performed on device drivers in the Linux kernel source tree [23] showed that the changes required could range from simple refactorings [9], such as renaming a function or reorganizing a data structure, to scattered changes across a function or file, potentially depending on some properties of the code context. Our goal was thus to devise a transformation language that would be easy for Linux kernel developers to use in automating all such changes.

When we started this investigation, the idea of using tools to generate, analyze, and transform systems code was already attracting interest. The Devil domain-specific language for generating kernel-device interface code from high-level specifications was proposed in 2000 [19], with a follow-up paper in 2001 investigating the robustness of the generated code [26]. Metal [7] was proposed in 2000 for systematically searching Linux and OpenBSD code for patterns of code corresponding to bugs, and was followed up with strategies for inferring such patterns from analysis of a code base [8, 11]. AspectC [5] appeared in 2001, targeting the instrumentation of FreeBSD with pluggable cross-cutting strategies for prefetching data from the disk. CIL [20], published in 2002, offered a parser and visitor for processing C code facilitating the development of analyses and transformations of C code at the abstract-syntax tree level. Some of these approaches, such as Metal and CIL, allow finding complex patterns of code and, in the case of CIL, transforming them. But they require programming-language-specific expertise, in automata and abstract-syntax trees, that are not within the comfort zone of the typical Linux kernel developer. None of these approaches enables systems code developers to easily create fine-grained transformation rules for the specific needs of porting Linux kernel code from an older version to a more recent one. Instead, it was Eelco Visser's Stratego/XT [1], a transformation system designed for general-purpose code written in languages such as Java, that provided a guiding light.

The rest of this paper illustrates some recurring code changes that were performed in Linux 2.5 and Linux 2.6, gives a brief overview of Stratego/XT and how it inspired our work, and concludes with a short presentation of our transformation tool for C code, Coccinelle [22]. Coccinelle was first released in open source in 2008. As of October 2022, Coccinelle has been used in over 9000 commits to the Linux kernel. More information about Coccinelle is available in a range of research papers on its design and application [3, 13, 14, 15, 16, 18, 24, 27], and at the Coccinelle web site.[1]

2 Some examples of Linux kernel changes

This section presents some widespread changes that were performed in Linux 2.5 or 2.6 that illustrate some important requirements for Coccinelle. The examples come from the `history` tree of the Linux kernel.[2] The change extracts, expressed as Unix *patches* [17], have been simplified for readability and to focus on the most relevant changes.

2.1 A new argument for `end_request`

Our first example illustrates a straightforward change that only requires considering a single function call. Nevertheless, it illustrates what can go wrong when developers make such changes manually or using tools such as search and replace that are not aware of the programming-language syntax.

Prior to Linux 2.5.22, the function `end_request` always worked on the request stored in the global variable `CURRENT`. In Linux 2.5.22, with the goal of eliminating `CURRENT`, the function was reorganized to take the request as an argument. The first step in the reorganization was thus to add `CURRENT` as a first argument to the existing calls, as illustrated by the patch in Figure 1.

```
@@ -929 +929 @@
- end_request(res);
+ end_request(CURRENT, res);
```

Figure 1 Change to `end_request` in `drivers/mtd/nftlcore.c` in commit 4fe6433a5d9e.

[1] https://coccinelle.gitlabpages.inria.fr/website

[2] git://git.kernel.org/pub/scm/linux/kernel/git/history/history.git

This straightforward change affected 40 files. Still, it was noted in the original work on Coccinelle [22] that the change was also incorrectly applied to a call to the unrelated function `swimiop_send_request`, showing the importance of a change-automation tool being aware of token boundaries in the programming language.

2.2 Changes in the usage of `usb_register_dev` and `usb_deregister_dev`

Our second example illustrates a case where constructing the changed code requires extracting information from other parts of the affected file.

In Linux 2.5.25, the function `usb_register_dev` lost its first argument and gained two new arguments, as illustrated by the patch in Figure 2. The new values are not part of the original call, nor in the adjacent code. Instead, they are the values stored in some members of the structure whose value was the original first argument.

```
@@ -815 +815,2 @@
- retval = usb_register_dev(&usblp_driver, 1, &usblp->minor);
+ retval = usb_register_dev(&usblp_fops, USBLP_MINOR_BASE, 1,
+                           &usblp->minor);
```

Figure 2 Change to `usb_register_dev` in `drivers/usb/class/printer.c`, in commit 26f8beab467e.

At the same time, the function `usb_deregister_dev` lost its first argument, as illustrated by the patch in Figure 3. This change is straightforward, requiring only information that is part of the original function call.

```
@@ -375 +375 @@
- usb_deregister_dev (&usblp_driver, 1, usblp->minor);
+ usb_deregister_dev (1, usblp->minor);
```

Figure 3 Change to `usb_deregister_dev` in `drivers/usb/class/printer.c`, in commit 26f8beab467e.

These changes affected 9 Linux kernel files, and show the importance of being able to collect information from across the definitions in a file, or even the entire code base.

2.3 Introduction of `kzalloc`

Our final example illustrates a case in which program control-flow has to be taken into account.

In Linux version 2.6.14, the function `kzalloc` was introduced, to replace the composition of a call to the Linux kernel memory allocator `kmalloc` and a zeroing call to `memset`, as illustrated by the patch in Figure 4.[3] Unlike our other examples, this change is not obligatory, because the functions `kmalloc` and `memset` remain unchanged in the Linux kernel. Still, making the change results in code that is more concise and readable, since the zeroing is immediately apparent at the allocation site.

[3] This change was overlooked over the years, and was made by the author in 2022: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3c0e3ca6028b`

```
- request = kmalloc(16, gfp_mask);
+ request = kzalloc(16, gfp_mask);
  if (!request)
    return -ENOMEM;
  urb = usb_alloc_urb(0, gfp_mask);
  if (!urb) {
    kfree(request);
    return -ENOMEM;
  }
- memset(request, 0, 16);
```

Figure 4 Introduction of `kzalloc` in `drivers/usb/net/zd1201.c`, in commit 3c0e3ca6028b.

This change raises a number of challenges. Commonly, `kmalloc` and `memset` are only separated by a conditional checking whether the allocation failed. Other intervening code, however, is possible, as shown in Figure 4, raising the need to adapt to different scenarios. Both `kmalloc` and `memset` are very common in the Linux kernel; the change should only be performed if the `memset` occurs in the control flow whenever the allocation succeeds and before the allocated memory is used. Furthermore, it is desirable for the amount of memory that is allocated and zeroed to be the same, as will be the case after the introduction of `kzalloc`.

3 Stratego/XT

Stratego/XT provides a language and a toolset for program transformation [1]. It grew out of Visser's work on rewriting in the late 1990s [31, 32]. Stratego/XT offers the ability to write complex transformations as a composition of a *strategy*, which is used to select terms to transform, and one or more transformation rules, which describe what changes to perform. Strategies describe how to navigate around an abstract-syntax tree to identify terms to consider for transformation. Transformation rules describe how to decompose an existing term, and use the components to build a new one. Stratego/XT is programming-language-independent. Internally, it adopts the *Aterms* of van den Brand et al. [28], that consists of a constructor name and a sequence of arguments. Stratego/XT uses ATerms to describe an abstract-syntax tree.

Writing transformation rules in terms of abstract-syntax trees requires the user to be aware of the constructor names and corresponding arguments that describe the various kinds of terms of interest. Obtaining this information may require studying documentation or even a language implementation, and the resulting transformation rules are verbose, since, e.g., even representing a simple variable `x` requires both the constructor for variable references and that constructor's argument list. The Stratego/XT user, however, does not interact with ATerms directly. Instead, Stratego/XT uses Syntax Definition Formalism (SDF), developed in Eelco Visser's PhD thesis [30], to provide parsers for various languages. A transformation is expressed as a fragment of concrete syntax, parameterized by *metavariables*, that is to be transformed into another fragment of concrete syntax, possibly referencing the metavariables mentioned in the former fragment. A metavariable, as previously proposed by van Deursen et al. [29], is a variable representing an arbitrary term of a particular syntactic category: expression, integer constant, etc. Metavariables are declared by prepending `~` to the variable name, but there are some predefined metavariables, such as `e` for an expression, or `i` for an integer constant. Using these predefined metavariables makes the fragments in the transformation rule look entirely like ordinary code, which makes the transformation to be performed instantly recognizable to the software developer.

At the time when we were first designing Coccinelle, Stratego/XT mainly supported Java code; a port for C code existed, but feedback from Eelco Visser suggested that it did not support enough of the C language to be reliably applied to the Linux kernel. In some experiments that we had performed in the context of another project, using Java, we also found that the strategies, while elegant, could also be complex to use, perhaps requiring more intuition about code as an abstract-syntax tree than would be typical of most Linux kernel developers. Nevertheless, the combination of fragments of concrete syntax, abstracted by metavariables, particularly metavariables that do not require any specific annotation within the fragments, appeared to be a powerful approach around which to design a user-friendly transformation language for real world software.

4 Coccinelle

Inspired by the ease of specifying transformation rules in Stratego/XT in terms of concrete syntax, in 2005, we began the design of Coccinelle. The initial idea was to propose a transformation language in which rules were explicitly composed of a phase for the collection of information from the source code, followed by a phase for the processing of that information. This design was motivated by Stratego/XT's strategies. Our postdoc, Yoann Padioleau, however, pointed out that this design still relied too much on a programming-languages point of view. To really provide something that would be usable by Linux kernel developers, we should directly follow their existing habits. Linux kernel developers are focused on lines of source code, and exchange and reason about changes in terms of *patches* [17], i.e., extracts of the source code in which lines to remove are indicated by a `-` at the beginning of the line, and lines to add are indicated by a `+`. Patches are easy to create from source-code changes, using the Unix `diff` tool [17], and easy to understand, because they contain the source code that the developer is already familiar with. They have the disadvantage, though, that each patch is completely tied to specific locations in the source code, due to the use of file names, line numbers, and the specific variable names, whitespace, etc. in the changed code and its context. To create Coccinelle's *semantic patch language*, SmPL, we considered how to make the patch notation more generic.

A semantic patch consists of a sequence of one or more *rules*, each of which describes code to remove and add. A SmPL rule has two parts: first a list of metavariables, surrounded by a pair of `@@`, that can be used in the second part, a pattern, describing the code to match and transform. Rather than using `~` and reserved names, as in Stratego/XT, SmPL metavariables are declared explicitly with their syntactic categories, such as `expression` or `statement`,[4] indicating the kind of code term that they can match. A pattern is a code fragment parameterized by metavariables. Lines in this fragment can be annotated with `-` and `+`, indicating code to remove and add, respectively. SmPL contains operators for describing (intraprocedural) program control-flow and expression types, i.e., some minimal semantic properties. We illustrate the main features of SmPL by revisiting the examples of Section 2.

[4] As fans of statically typed languages, we considered such declarations to be important, but we were told later by some systems code developers with no formal programming-languages training that they found the need to choose the syntactic category of a metavariable to be a burden, because they were not familiar with the terminology. We added a generic syntactic category `metavariable`, but this syntactic category cannot always be used, because the metavariable syntactic categories are sometimes essential to avoid parser conflicts.

4.1 A new argument for `end_request`

To make the change for `end_request` (Figure 1), it is necessary to match the existing call, remove it, and replace it with one that has `CURRENT` as an extra first argument. The form of the original argument is not important, and so we abstract over this argument with a metavariable. Note that the resulting pattern is identical to the code found in the example change in Figure 1. As the argument was already a simple variable in the code of Figure 1, it suffices to abstract over this variable.

```
@@ expression res; @@
- end_request(res);
+ end_request(CURRENT, res);
```

Some variations on the above semantic patch are possible. First, a function call, such as the call to `end_request`, need not appear at the top level in a statement. To match the call as an arbitrary subexpression, the trailing semicolon can be omitted. The resulting semantic patch will match a function call expression, wherever it occurs, rather than only as a complete statement. Second, in the spirit of the traditional patch syntax, Coccinelle allows adding and removing any sequence of tokens, as long as the pattern to match against the existing code and the pattern to create the generated code each represents a well-formed term in the C language (statement, expression, type, etc.). Thus, rather than removing the entire `end_request` call, we can leave the existing code in place and simply add a new first argument. A semantic patch incorporating both variations is as follows:

```
@@ expression res; @@
 end_request(
+  CURRENT,
   res)
```

In either case, there is no danger of matching a call to `swimiop_send_request`, as Coccinelle tokenizes and parses the code according to the grammar of the C language.

4.2 Changes in the usage of `usb_register_dev` and `usb_deregister_dev`

To make the change for `usb_register_dev` (Figure 2), it is necessary to find the values of certain members of the structure that is the first argument of the call to `usb_register_dev`. Coccinelle processes each top-level declaration (function, variable, type, etc.) separately, so matching both the structure definition and the call to `usb_register_dev` requires multiple rules. We construct a semantic patch that first matches the structure and then uses the collected information to update any call to `usb_register_dev` that refers to that structure. While most SmPL patterns, as illustrated in Section 4.1, only match code that has the exact form presented, there is an exception in the case of structure initializations. As the C language only requires specifying values for the non-zero members and allows the member values to be specified in any order, the first rule, below, matches any structure initialization that provides values for at least the members `fops` and `minor`, in any order. We give this first semantic patch rule the name `rule1` (line 1), so that the metavariable bindings that it creates can be referred to in subsequent rules.

```
1 @ rule1 @
2 identifier I;
3 expression fops_val, minor_val;
4 @@
```

```
struct usb_driver I = {
        fops:           fops_val,
        minor:          minor_val,
};
```

Equipped with the information stored in the `fops` and `minor` members of the `usb_driver` structure, we then create a second rule to update the call to `usb_register_dev`. This rule *inherits* the various metavariables bound by `rule1` (lines 2 and 3). If a file initializes multiple `usb_driver` structures, this rule will be applied once per distinct set of bindings for the inherited metavariables.

```
1 @@
2 identifier rule1.I;
3 expression rule1.fops_val, rule1.minor_val, E1, E2;
4 @@
5    usb_register_dev(
6 -    &I
7 +    fops_val, minor_val
8      , E1, E2)
```

The transformation for `usb_deregister_dev`, shown below, is then straightforward. Note that the rule could have been written to simply remove the first argument, regardless of its form. However, the following rule checks that this argument is the same as the name of the matched structure. In this way, the transformation will only be carried out if it can be carried out for both the `usb_register_dev` and `usb_deregister_dev` calls. If the needed information is not available, both calls will remain with the wrong number of arguments, having the wrong type. Untransformed code will thus cause an error when compiled, signaling to the user that the semantic patch has to be extended to take some other conditions into account.

```
@@
expression E1, E2;
identifier rule1.I;
@@
  usb_deregister_dev(
-                    &I,
                     E1, E2);
```

4.3 Introduction of `kzalloc`

To introduce the use of `kzalloc`, it is necessary to find a call to `kmalloc` such that the entire allocated region is subsequently zeroed using `memset`. As illustrated by the example patch in Figure 4, the calls are not necessarily adjacent, and indeed they may be separated by some code that is specific to the local context. Instead, we need to ensure that every execution that passes through the call to `kmalloc` also includes execution of a call to `memset`. To describe such control-flow paths, Coccinelle provides the pattern element "`...`".

A simple semantic patch introducing `kzalloc` is shown below:

```
@@
expression x, size, flag;
@@
- x = kmalloc(size, flag);
+ x = kzalloc(size, flag);
  ...
- memset(x, 0, size);
```

This semantic patch ensures that the return value of kmalloc is stored in an expression that has the same form as the expression in the first argument of memset, and that the second argument of kmalloc is an expression that has the same form as the expression in the third argument of memset. It also ensures, via the "...", that every execution path through the call to kmalloc that does not end up in a failure case, e.g., when x is detected to be NULL, leads to the call to memset. Coccinelle marks failure cases at parse time, by searching for if statements with only one branch, where the branch ends in a return or goto, which is a typical coding practice in the Linux kernel. The use of "..." furthermore ensures that the matched code does not contain another identical call to kmalloc or to memset, as "..." matches the shortest possible execution path between two terms matching the provided patterns.

On the other hand, the above semantic patch does not ensure full semantic correctness, e.g., that x or size is not redefined between the call to kmalloc and the call to memset. To check for unwanted code within an execution path, Coccinelle allows "..." to be annotated with when clauses indicating patterns of code that should not occur in the matched region. Furthermore, the above semantic patch does not accommodate the case where the size of the allocated region is expressed in different ways in the calls to kmalloc and memset, e.g., using the size of the type of x in the kmalloc call and the size of *x in the memset call. To address this issue, more rules can be used. These features are illustrated in a recent paper presenting some uses of Coccinelle in more depth [15].

Overall, Coccinelle aims for a WYSIWYG approach (or WYSIWIB – "what you see is where it bugs", in the case of semantic patches that are designed to find and possibly fix bugs [16]). A developer can write a semantic patch that corresponds to the developer's understanding of the code and the issues that should be taken into account. The developer can then test the resulting semantic patch on all or part of the Linux kernel, examine the results obtained, and extend the semantic patch if needed, to make it more defensive in the case of false positives, and to take into account more cases if some variations seem to be overlooked. Coccinelle provides very minimal syntactic safety, e.g., ensuring that an expression is replaced by another expression, but otherwise the developer can iteratively work towards a semantic patch that is as safe as necessary. Through this process, the semantic patch remains readable due to the use of patterns of C code that are laid out in a way that reflects how they appear in the program source code.

5 Implementation

The primary focus of this paper is on the user experience with Stratego/XT and with Coccinelle. Nevertheless, to achieve a user experience, it is necessary to actually implement the transformation system.

Coccinelle is implemented in OCaml. It relies on two parsers, one for SmPL and one for the C language. Both parsers were written from scratch, using the yacc-like parser generators menhir [25] and ocamlyacc, respectively. Creating these parsers from scratch allowed collecting exactly the information required by Coccinelle, including information about comments and whitespace; such information is traditionally discarded by compilers, but is essential to generate transformed source code that is identical to the original source code except at the places where transformations were performed. Creating a parser from scratch also allowed treating the exact variant of the C language used by the Linux kernel, made it possible to avoid macro expansion by parsing the macro uses directly, which both offers efficiency and makes it possible to match and transform macro uses [14], and enabled creating a parser for SmPL that can accept arbitrary code fragments, notably expressions, that are not designed to occur at the top level in the C grammar. More details about the parser design are presented by Padioleau [21]. The main components of the rest of the implementation of Coccinelle include an encoding of SmPL in a variant of computational tree logic (CTL) [3], inspired by the work of Lacey et al. on proving the correctness of compiler optimizations using temporal logic [12], and a custom pretty printer that reconstructs the program from the original tokens, modified by the specified transformations. In practice, it is the pretty printer that has required the greatest maintenance effort over the years.

After completing Stratego/XT, Visser designed the language workbench, Spoofax [10], with the goal of facilitating the design and implementation of domain-specific languages (DSLs), including generation of syntactic and semantic verifications and generation of an associated integrated development environment (IDE). As Coccinelle is essentially a domain-specific language for transforming C code, we may consider whether a tool such as Spoofax would have facilitated its development. While Spoofax includes many impressive features, it turns out to be a poor match for Coccinelle. A main goal of Coccinelle is to fit with the habits of the Linux developer community. It is traditional in this community to use command line tools and to not impose any particular graphical user interfaces for code development and maintenance. To fit with these traditions, Coccinelle was specifically designed as a command-line tool based on text files, that users were free to create however they wanted. In terms of parsing, most of the challenge, whether for the SmPL parser or for the C parser, is to parse C code, and to do this without expanding macros. The C language is not context free, and the goal of not expanding macros adds further complexity. The parser design proposed by Padioleau to cope with this challenge [21] involves arbitrary lookahead between the lexer and the parser, and re-encoding the tokens according to the collected information. This kind of flexibility is not offered by Spoofax, and is probably not necessary for the kind of well designed DSL that Spoofax targets. Furthermore, the type inference and name resolution offered by Spoofax are of limited use for Coccinelle, as SmPL does not offer nested scopes.

Still, existing language infrastructure can be useful in implementing variants of Coccinelle for some languages. Recently, we have started developing a variant of Coccinelle targeting Rust. For this, we are reusing the parser and abstract-syntax tree offered by Rust Analyzer [6]. Like C, Rust offers macros, but macro uses must conform to the syntax of the Rust language. Rust Analyzer, unlike the off-the-shelf parsers existing for C at the time when we developed Coccinelle, also retains all whitespace and comments. While Coccinelle for Rust remains work in progress, we hope that the use of Rust Analyzer will lead to a simpler implementation.

6 Conclusion

Coccinelle has been under development for more than 15 years. It is used extensively in the Linux kernel and in other C software. It continues to evolve, to better support the C code as found in the Linux kernel (a recent request was for better parsing of Sparse attributes [2]), and to support similar programming languages, such as C++. At the heart of the Coccinelle philosophy is the goal of doing one thing and doing it well, in a tool that is usable in practice by domain experts. A similar spirit also infused Eelco Visser's work.

References

1 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

2 Neil Brown. Sparse: a look under the hood. *Linux Weekly News*, 2016. https://lwn.net/Articles/689907/.

3 Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

4 Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

5 Yvonne Coady, Gregor Kiczales, Michael J. Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In A Min Tjoa and Volker Gruhn, editors, *ESEC/FSE*, pages 88–98, 2001.

6 Ferrous Systems & contributors. rust.analyzer, 2023. `https://rust-analyzer.github.io/`.

7 Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.

8 Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

9 Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2002.

10 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Object oriented programming systems languages and applications (OOPSLA)*, pages 444–463, October 201.

11 Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson R. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, pages 161–176, 2006.

12 David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Principles of Programming Languages (POPL)*, pages 283–294. ACM, 2002.

13 Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding error handling bugs in OpenSSL using Coccinelle. In *EDCC*, pages 191–196, 2010.

14 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*, pages 601–614, 2018.

15 Julia Lawall and Gilles Muller. Automating program transformation with Coccinelle. In *NASA Formal Methods (invited talk)*, volume 13260 of *Lecture Notes in Computer Science*, pages 71–87, 2022.

16 Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. *Software: Practice and Experience*, 43(1):67–92, 2013.

17 David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch.* Network Theory Ltd, January 2003.

18 Michele Martone and Julia Lawall. Refactoring for performance with semantic patching: Case study with recipes. In *High Performance Computing - ISC High Performance Digital 2021 International Workshops*, volume 12761 of *Lecture Notes in Computer Science*, pages 226–232, 2021.

19 Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *OSDI*, pages 17–30, 2000.

20 George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, 2002.

21 Yoann Padioleau. Parsing C/C++ code without pre-processing. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2009.

22 Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

23 Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys*, pages 59–71, 2006.

24 Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. Faults in Linux 2.6. *ACM Transactions on Computer Systems*, 32(2):4:1–4:40, 2014.

25 François Pottier and Yann Régis-Gianas. Menhir reference manual, February 2022. `http://gallium.inria.fr/~fpottier/menhir/`.

26 Laurent Réveillère and Gilles Muller. Improving driver robustness: An evaluation of the Devil approach. In *DSN*, pages 131–140, 2001.

27 Luis R. Rodriguez and Julia Lawall. Increasing automation in the backporting of Linux drivers using Coccinelle. In *EDCC*, pages 132–143, 2015.

28 Mark van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

29 Arie van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

30 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

31 Eelco Visser and Zine-El-Abidine Benaissa. A core language for rewriting. In *International Workshop on Rewriting Logic and its Applications, (WRLA)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 422–441. Elsevier, 1998.

32 Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.

33 Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

Eelco Visser as a Founding Member of the IFIP WG 2.11

Christian Lengauer
Universität Passau, Germany

Jacques Carette
McMaster University, Hamilton, Canada

Abstract

Appreciation of Eelco Visser's contribution to the IFIP WG 2.11 by two of its chairs. Christian Lengauer was chair from 2007 to 2013. Jacques Carette has been chair since 2019.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases IFIP WG 2.11

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.19

Acknowledgements Thanks to Schloss Dagstuhl for permitting us to reproduce the image in Figure 1 at Web link `https://www.dagstuhl.de/storage/media/0000/709/conversions/03131.01.l.jpg`.

Eelco's Impact on the Working Group

The sudden, untimely and entirely unexpected death of Prof. Eelco Visser was a severe shock to the members of IFIP WG 2.11. The group lost one of its founding members, a faithful participant and central player in the group for close to two decades.

The IFIP WG 2.11 has been a cheerful and lively group from its inception. Eelco contributed to this spirit continuously. He stood out by his calm demeanor and pleasant, sonoric voice whilst clearly being extremely passionate about the group's work. It gave him a special presence. Some time ago, he told us that he considered his first invitation a pivotal event in his career for which he was very grateful. This was Dagstuhl Seminar 03131 in 2003, which can be viewed as the conception event of the WG 2.11 as its formation was decided there.

Eelco implemented the group's goals more effectively and tangibly than many of us and he never made a big deal about it. The software systems he developed, among the more widely known Stratego [3], with which he introduced himself to our group, Spoofax [1], and researchr [2], which is based in his web engineering platform WebDSL [4], have had a profound effect onto our community and beyond. His work managed to cross domains quite successfully (chiefly programming languages and software engineering) while also ranging from rather theoretical to eminently practical. When Eelco built theory, he built it with a clear purpose in mind.

Eelco often used the working group meetings as a sounding board for some of his latest ideas (see his talk titles below) and got plenty of feedback. He trusted the members of the group to be appropriately critical, in the original sense of "critique" which involves a careful weighing of pros and cons. Others have emulated Eelco in this manner.

Eelco was always courteous, attentive and friendly but, being also passionate, he would not shy away from lively arguments. He was a pillar of strength and steadiness in the group. He will be dearly missed.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 19; pp. 19:1–19:3
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Eelco's Presentations at WG 2.11 Meetings

- 05/2004, St. Emilion (France): present but no talk recorded
- 03/2005, Houston (USA): "Regenerative Programming: A New Approach to Data Mining for Software Design"
- 01/2006, Schloss Dagstuhl (Germany): "A Framework for Transformation of Java Programs"
- 10/2006, Portland (USA): Session Chair "Teaching Program Generation"
- 08/2007, Copenhagen (Denmark): "Domain-Specific Language Engineering. A Case Study in Agile DSL Development"
- 06/2008, Passau (Germany): "Code Generation by Model Transformation"
- 03/2010, St. Andrews (UK): "Building IDEs for Domain-Specific Languages with Spoofax/IMP"
- 09/2011, Bordeaux (France): "Declaratively Programming the Mobile Web with Mobl"
- 06/2012, Halmstad (Sweden): "Declarative Language Definition (in Spoofax)"
- 01/2015, Stellenbosch (South Africa): "A Theory of Name Resolution"
- 08/2016, Bloomington (USA): "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics"
- 06/2018, Kyoto (Japan): "Declarative Disambiguation of Deep Priority Conflicts"
- 04-05/2019, Salem (USA): "Declarative Type System Specification with Statix"
- 02/2020, Paris (France): "Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System"

References

1 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *ACM SIGPLAN Notices*, 45(10):444–463, October 2010. OOPSLA 2010. Most Influential Paper Award at OOPSLA 2020.

2 Elmer van Chastelet, Eelco Visser, and Craig Anslow. Conf.researchr.org: Towards a domain-specific content management system for managing large conference websites. In Jonathan Aldrich and Patrick Eugster, editors, *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications (SPLASH): Software for Humanity*, pages 50–51. ACM, 2015.

3 Eelco Visser. Program transformation with Stratego/XT. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, LNCS 3016, pages 216–238. Springer-Verlag, 2004.

4 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, LNCS 5235. Springer, July 2008. GTTSE 2007.

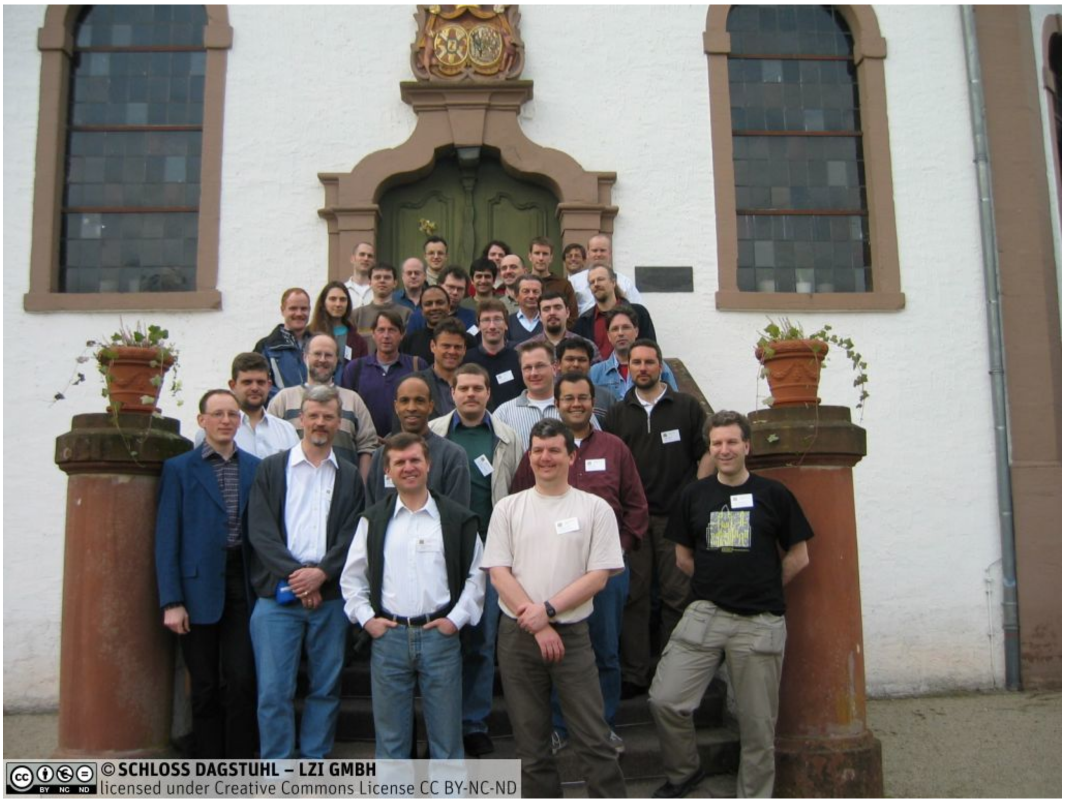


Figure 1 The participants of IFIP WG 2.11's conception seminar at Schloss Dagstuhl. Eelco is in the second row from the bottom at the far right, the first author is directly behind him.

Visitor Optimization Revisited – Realizing Traversal Graph Pruning by Runtime Bytecode Generation

Markus Lepper
semantics gGmbH Berlin, Germany

Baltasar Trancón y Widemann
Nordakademie Elmshorn, Germany
semantics gGmbH Berlin, Germany

Abstract

Visitors and Rewriters are a well-known and powerful design pattern for processing regular data structures in a declarative way, while still writing imperative code. The authors' "umod" model generator creates Java data models from a concise and algebraic notation, including code for visitor skeleton classes according to traversal annotations. User visitors are derived from these, overriding selected generated methods with payload code. All branches of the visiting trajectory that are not affected can thus be safely pruned according to control flow analysis. In the first version [7], the pruning was implemented by dynamic case distinction. Here we have developed a new solution employing code generation at runtime.

2012 ACM Subject Classification Software and its engineering → Software performance; Mathematics of computing → Paths and connectivity problems; Theory of computation → Program analysis

Keywords and phrases Visitor Pattern, Generative Programming, Control Flow Analysis, Reflection, Runtime Code Generation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.20

Supplementary Material *Software (Source Code)*: `http://bandm.eu`

1 Introduction, Context

Visitors and Rewriters are a well-known and powerful design pattern for processing regular data structures in a declarative way, still writing imperative code. See Gamma et al. [4], Palsberg et al. [13], and VanDrunen et al. [18] for the foundations; see Nanthaamornphong et al. [11] for an empirical study encouraging their use, and for a survey on such studies. Pati and Hill [14] give a survey on extensions and alternatives of the original pattern – the umod approach as described below would classify generally as their "dynamic type" and "acyclic", its multiphase variant as a "hierarchical" one. Petrashko et al. [15] present a full-fledged compiler architecture as a sequence of visitor/rewriter phases and emphasize the optimization needs for compilers in practical engineering. Taking a bird's-eye view upon complete model definitions, and planning and optimizing traversals by algebraic means has been a main topic in "Adaptive Programming" [3][9].

The authors' "umod" model generator creates Java data models from a concise and algebraic notation [7]. (For more related work prior to 2011 please refer here.) The tool generates source text files for the classes which realize the elements of the model. The factor between the lines of codes of an umod source and the generated Java is in the range of 1:25 to 1:40. Such a reduction significantly improves order, robustness, and maintainability in software projects, and is a strong argument in favour of source text generation.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 20; pp. 20:1–20:12
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
 1 MODEL M =
 2    VISITOR 0 Simple ;
 3    VISITOR 1 Rewriter IS REWRITER ;
 4    VISITOR 2 Visitor2 MULTIPHASE ;
 5
 6 TOPLEVEL CLASS
 7   A
 8       a1  int <-> B1 ! V 0/0 1/0 2/0 ;
 9       a2  A          ! V         2/1 ;
10   | B1
11       b1  OPT A      ! V     1/0     ;
12       b1b SEQ B1     ! V 0/0         ;
13   | B2
14       b2  int -> D   ! V 0/0 1/0 2/0 ;
15       b2b OPT B2     ! V 0/1 1/1     ;
16   D
17       d   int  = "17"
18   | E
19 END MODEL
```

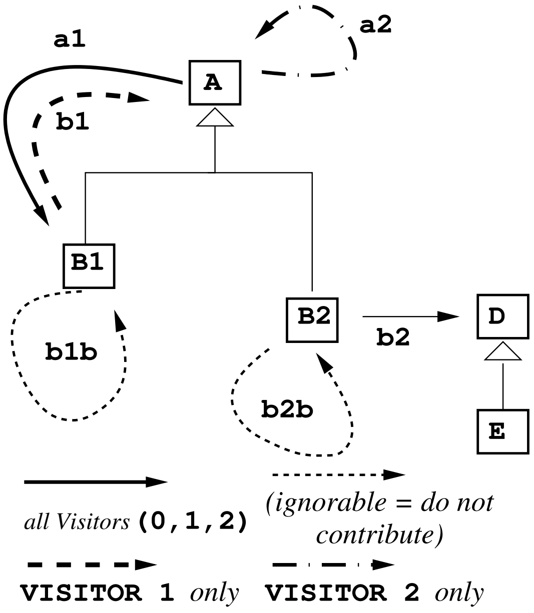


Figure 1 A simple example model definition.

Models realized by umod are collections of instances of these classes, called *model elements*, created one after the other explicitly by the programmer, with host language constructor calls, and linked together by field values. (For technical details please refer to the original publication [7].) Umod is currently the basis for about twenty small to medium-scale projects in academy and industry.

Figure 1 shows an example: Uppercase identifiers are names of class declarations, lower case are instance fields. Indentation with the "`|` " operator indicates inheritance/subclassing.

The types of fields are primitive types, references to model classes, references to external classes, or free applications of the constructors `SEQ`, `MAP/->`, `REL/<->`, `OPT` and tupling/`*`. These behave fully compositionally, allowing complex nestings like "`((SEQ int) * string) <-> (int -> OPT int)`". The generated code includes safe constructor and setter methods that reject spurious `null` values, serialization, visualisation, etc.

Much of the power of umod comes with the generated code for visitors and rewriters. in the second halves of the source lines 8, 9, 11, 12, 14, and 15 in Figure 1, the annotations of form "`V` *a*/*b*" define *traversal plans*: The value of *a* is a numeric identifier of such a plan, the order of the numbers *b* gives the sequential order of visiting the fields of this class.

Umod models are not restricted to tree shape – their fields with reference values can span arbitrary graphs, possibly including cycles. Any straightforward solution like the classical `Walkabout` [13] cannot cope with these. The explicit selection of the fields to be followed allows for cycle-free traversal plans (= "spanning trees"), much more efficiently implemented.[1]

Using these plans, the generation of visitor classes of different types (plain or multiphase, rewriters, cyclic rewriters, visualizers etc.) can be declared, as shown in lines 2–4 of the Figure.

[1] Theory and implementation of cyclic visitors and rewriters is much more challenging, see [8].

2 Optimization of Visitor Traversal

As known from other visitor frameworks, the visitor pattern is employed by (A) deriving a class from the abstract visitor's base class (here: source generated by umod), (B) overriding a dedicated, overloaded `visit` method for some selected classes with payload code, and finally (C) invoking some top-level `visit` method with the object (graph/model) to be processed.

The `visit` methods of the generated base class perform just the traversal, according to the traversal plan. They can explicitly be called from the payload code where appropriate.

When campaigning to introduce generative tools into practice, the gains in production speed and maintainance always stand against a (presumed or real) loss in execution speed. Therefore all potentials for *optimization* should be explored. All descending calls which never (transitively) reach a payload method, according to the selected traversal plan and the overridings in this particular programemr-defined visitor subclass, can thus be pruned for optimization.

The overridden payload methods are inherited along two axes: first by a visitor from its superclass, then (by explicit casting in the generated code) among the visitees. Descending into one class valued field successively follows the traversal plan of all ancestors of this class, and potentially any of its descendant classes. The right side of Figure 1 shows the UML graph with the associations selected by the different traversal plans.

Must a request to visit an instance of class `B1` really be satisfied when there is only one single payload definition for class `E`? Not with traversal plan `0`. But in plan `1` each `B1` has a direct reference to an object of type `A` (in UML terms: an “association”), which can be an instance of `B2`, which has an association to `D`, and any instance of this could be an `E`. In plan `2` such a reference to `B2` is even more indirect, namely inherited by `B1` from `A`.

The *pruning condition* states for every field of every class in the umod definition whether descending into its value can possibly reach a payload method. It is calculated for one particular programmer-defined visitor by combining (a) the subclass relation of the model classes, (b) the map from all field definitions to the sets of all model classes which occur in their type, (c) the set of fields selected by the traversal plan, and (d) the set of model classes with an overridden visit method. (a)–(c) are explicit in the model definition, and hence known to umod when generating the Java source; (d) is specific to the application-defined visitor class, and extracted at runtime from the loaded class by means of *reflection*.

The expressions for $\mathsf{classGuard}_U$ and $\mathsf{fieldGuard}_U$ in Figure 2 give the the pruning condition per class or per field, resp., as needed for the two variants of the optimization. Most of it can be calculated at model compilation time. An intermediate step is to condense the traversal graph into strongly connected components (SCCs) of classes, and all further graph analysis is performed on these rather than individual model classes.

In the original implementation [7], the class loading code generated an array of boolean flags indexed by field ids, and the generated visitor code contained an explicit test before descending into the current value of a particular field – see the value of $\mathsf{fieldGuard}_U$ in Figure 2. This variant is called AOT in the following, because all code is created *ahead of time*.

A first result of this project was an empirical performance improvement in the range of 10–20% for a realistic application.

A second result was a methodological warning to software developers: The analysis of the reachability relation turned out to be quite more complex than expected. For example: The associations from a class to itself `B1.b1b` and `B2.b2b` do not contribute to the set of reachable classes and can be ignored, see Figure 1. But `A.a2` does, because of subclassing. Thus any attempt to apply traversal pruning “manually”, when coding, is not advisable:

$$
\begin{array}{ll}
C & \textit{// = all model classes} \\
F & \textit{// = all fields, by ids unique across all classes} \\
\mathsf{super} : C \nrightarrow C & \textit{// maps class to its superclass, if not a root class} \\
\mathsf{fieldOf} : F \to C & \textit{// maps field to containing class} \\
\mathsf{refersTo} : F \leftrightarrow C & \textit{// links fields to the model classes appearing in its type} \\
\mathsf{select}_n : 1 \leftrightarrow F & \textit{// = fields selected by the traversal plan } n \\
\multicolumn{2}{l}{\mathsf{subSup} : C \leftrightarrow C = \mathsf{super}^* \cup \mathsf{super}^{\smile *}} \\
\multicolumn{2}{l}{\mathsf{reaches}_n : C \leftrightarrow C = (\mathsf{subSup} \fatsemi \mathsf{fieldOf}^{\smile} \fatsemi \mathrm{ID}_{\mathsf{select}_n} \fatsemi \mathsf{refersTo})^*} \\
\mathsf{scc}_n : C \to C & \textit{// each SCC is represented by one of its members} \\
\multicolumn{2}{l}{\mathsf{reaches}_n \cap \mathsf{reaches}_n^{\smile} = \mathsf{scc}_n \fatsemi \mathsf{scc}_n^{\smile}} \\
\mathsf{payload}_U : 1 \leftrightarrow C & \textit{// = for one particular programmer defined visitor } U\textit{: the set of} \\
 & \textit{//}\qquad \textit{all classes for which the visit method is overridden.} \\
 & \textit{// Let the underlying traversal plan of } U \textit{ be } p. \\
\multicolumn{2}{l}{\mathsf{classGuard}_U : 1 \leftrightarrow C = \mathsf{payload}_U \fatsemi \mathsf{subSup} \fatsemi \mathsf{scc}_p \fatsemi \mathsf{reaches}^{\smile} \fatsemi \mathsf{scc}_p^{\smile}} \\
\multicolumn{2}{l}{\mathsf{fieldGuard}_U : 1 \leftrightarrow F = \mathsf{classGuard}_U \fatsemi \mathsf{refersTo}^{\smile}} \\
\multicolumn{2}{l}{\textit{// \_} \fatsemi \textit{\_ is relational composition. \_}^{\smile} \textit{ is inversion. \_}^{*} \textit{ is reflexive–transitive closure}} \\
\multicolumn{2}{l}{\textit{// ID\_ is the identity restricted to a set. 1 is any singleton type, used for modelling sets.}}
\end{array}
$$

Figure 2 The umod visitor optimization, symbolically.

```
protected void action(E elem) {
  ...
  if (fieldGuard[idOfField]) {match(elem.field);}
  ...
}
```

Figure 3 Old implementation of pruning, AOT variant.

The strength of the visitor pattern is its declarative nature, which brings compositionality, maintainability, and automated adaption to model changes, all of which are jeopardized by manual intervention.

3 Pruning by Runtime Code Generation

In the original implementation (AOT), the pruning has been realized by explict guards on the caller side; see Figure 3. Although somewhat redundant, this appears to be a more reasonable choice for performance than placing guards on the callee side: it is not likely for the just-in-time compiler to inline a method that contains a potentially large conditional statement, and hence unnecessary calls would be expected.

More recently, we have been experimenting with the potential of dynamic program specialization by runtime bytecode generation for staged meta-programming [16, 17]. This allows for a distinct implementation strategy called JIT (because part of the code is created just in time, at runtime): In a subclass of the given programmer-defined visitor, methods that constitute entry points for unaffected SCCs can be overridden with code that returns immediately; see Figure 4. Modern JIT compilers can be trusted to inline such small method bodies aggressively, thereby fully eliminating the need for caller-side checks. As a bonus, contrarily to the AOT implementation, visitor optimization has zero cost, regarding both

```
// if (!classGuard[idOfS]) generate the following:
@Override
protected void action(S subElem) {
  return;
}
```

Figure 4 New JIT variant of pruning.

```
new MyVisitor() {
  ...
}.compile().match(rootElement);
```

Figure 5 Explicit use of specialized visitors.

code size and runtime performance, in cases where no pruning can be applied. Furthermore, the partial pruning of fields with complex types, such as `A -> B` where `B` is affected but `A` is not, is supported.

For this strategy to take effect, the visitor object that is actually invoked needs to be replaced by a quasi-clone, an instance of a suitably specialized subclass generated on demand by the umod runtime library. This is almost transparent to the programmer; see Figure 5: Every visitor class provides a method `compile()` that creates and returns such a quasi-clone. Code is generated only once per programmer-defined visitor class, and subsequently cached. If the compiler is switched off or otherwise not available, `compile()` falls back to simply returning the original, such that programmer code may work as intended, only with suboptimal performance.

The runtime support in the umod library consists essentially of the inherited base method `compile()`, and has been implemented in some 25 lines of code, using the LLJava-live code generator library [17]. LLJava-live provides a concise builder API and efficients generator for JVM bytecode to be loaded straightaway into the running application. All Java classes, including inner, local and anonymous classes can be subclassed.

For the umod specialization mechanism to work as expected, three preconditions must be met: First, the visitor class to be specialized must have an accessible constructor that takes no user-visible arguments. Second, necessary configuration with setters must be performed *after* compilation. Third, the Java 9+ module system precludes access to the captured variables of nested classes across class loaders. Hence, support for specialization of nested classes, as in Figure 5, requires the application to be loaded with a specific class loader provided by LLJava-live. In the next section this class loader is used also for the other variants, for fair comparison.

4 Evaluation

In order to compare the dynamic and the generator-based implementations of pruning with each other and the naïve version in detail, we have constructed some small benchmark tests. As the umod model for these experiments we use DTM, a semantic model of XML DTDs [2]. In DTM, parameter entities, cross-references between elements, attribute lists, and XML namespaces are resolved. The umod source defines 29 model classes in about 100 lines of code. All experiments operate on two fixed DTM document objects, namely the XHTML-1.0-Strict DTD [20] and the SVG-1.1 DTD [19] (978/5664 LoC in original form, respectively).

Table 1 Benchmark results (XHTML-1.0-Strict).

Case	Naïve	AOT	JIT
nop	40.06 μs ± 0.41%	0.05 μs ± 2.08%	0.10 μs ± 6.12%
nms	70.89 μs ± 0.85%	47.69 μs ± 0.86%	49.62 μs ± 1.05%
rep	37.00 μs ± 0.42%	21.64 μs ± 0.54%	7.94 μs ± 0.43%
req	39.23 μs ± 0.46%	21.67 μs ± 0.50%	20.67 μs ± 0.38%
few	43.11 μs ± 0.39%	0.96 μs ± 0.31%	1.78 μs ± 1.07%

Table 2 Benchmark results (SVG-1.1).

Case	Naïve	AOT	JIT
nop	70.97 μs ± 1.16%	0.06 μs ± 1.64%	0.42 μs ± 68.11%
nms	205.70 μs ± 5.07%	198.14 μs ± 2.94%	194.81 μs ± 4.22%
rep	87.40 μs ± 1.75%	67.44 μs ± 0.69%	7.60 μs ± 0.51%
req	86.10 μs ± 1.46%	71.37 μs ± 0.61%	74.36 μs ± 1.12%
few	95.74 μs ± 1.38%	1.09 μs ± 0.18%	2.05 μs ± 1.02%

Each benchmark case consists of a single visitor class that performs a simple query on the data, and hence overrides no more than two methods: *nop* does nothing at all; *nms* collects the set of all names in the document (167/340); *rep* counts the elements with repeatable content, i.e., which use the operator `+/*` (17/63); *req* counts the required attributes (13/40); *few* counts the elements with fewer than five attributes (3/4).

Running times are estimated as wallclock times, measured with `System.nanoTime()` precision. Each query is repeated $N = 50000$ times, after a JIT compiler warmup phase of the same length. A fresh visitor instance is created in each iteration, and its `compile()` method is invoked. The reported numbers are median values and median absolute deviations. All measurements have been performed on a system with a Core i5-10210U CPU, running Ubuntu 20.04 and OpenJDK 11.0.16.

The results indicate that the performance improvement by code generation are largely comparable with those by dynamic pruning, and that both can be quite dramatic for "sparse" visitors operating on stratified models, where a lot of pruning can occur. It is unclear how much just a more aggressive JIT compiler would improve the naive approach.

The instantiation of a runtime-generated visitor subclass in the JIT variant requires the use of reflection, and thus incurs some initial overhead; as can be seen in the *nop* case, the overhead is generally small but potentially erratic. By contrast, generated code performs significantly better in the *rep* case, where pruning occurs much more sparingly and at deeper levels of nesting than in the other cases.

5 Umod Visitors Meet Strategic Programming

Strategic Programing (SP), as developed by Eelco Visser and others, is in the first line a *concept*, comprising term definitions, requirements, categories, abstract algorithms, etc. In a second step this concept can be "incarnated" in programming languages of different paradigms: functional, object oriented, data driven, etc.; see Figure 7 in [6], and more in Figure 6 in [10].

Umod differs from most other approaches to visitor generation: It is a *domain-specific language (DSL)* with own syntax, semantics and implementation. But it is neither realized by genuine means of the host language (*Embedded DSL, EDSL*) as in the "Scrap your boilerplate" projects [3] [5] and the Scala "Miniphases" [15]. Nor are its syntax and that of the host language merged into the input format of a dedicated compiler, as with "Tom". [1] Instead, the DSL is processed totally independently from the programmer's Java source, by which its translation result finally will be called – in so far similar to other visitor generators [9], [12], and [18]. But umod does not only generate visitor code, but the complete class sources which make up the model definition, including constructors and setter methods, both with null check, visualisation, de/serialization, uncurrying and implicit creation of nested containers, etc.

The generated code of the different variants of visitors carries out the fundamental operations only: traversal and cycle detection anyhow, plus the tedious task of clone generation and management by the rewriters. All other functionality must be added by the programmers explicitly, by the host language means they are familiar with.

Nevertheless, the theory of Strategic Progamming can sensibly be applied also to the visitors of umod, for classification and clarification of its relations to other concepts. Applying the check list from [6], p. 171, yields:

Genericity Being Java class definitions, visitors can take part in the standard Java type parametrization, and can be constructed for models which are themselves parametric. (Both is currently not yet implemented, but does not impose fundamental problems.) A generic handling is *not* possible for *field names*. Nevertheless, inheritance, as discussed below, has turned out sufficient for adaptation and re-use of visitors with variants of models.

Specificity Since the activities are defined by genuine host language code operating on statically typed arguments, all their details are accessible.

Compositionality Being Java class definitions, umod visitors are *not* compositional in the strictest sense, as demanded for strategies. But they profit from the host language's *inheritance* rules along two axes: The generated code calls as default the visiting methods of the argument's superclass, and visitors can be derived from visitors. This allows their refinement, heavily employed in programming practice, see for instance Figure 6.[2]
Visitors are normal objects and thus first class citizens: they can be passed as arguments and used by other visitors. This is a weak form of compositionality, as usual in object oriented languages, see Figure 7.

Traversal The sequential order of the fields on one level of class definition can/must be specified explicitly by the user, see Figure 1. These sequences are concatenated starting with the most specific class upwards to the most general. The content of Java collections is visited by the standard iterators. For instances, the traversal plan `1` in Figure 1 will for each instance of `B2` first visit the objects of type `D` from the map in field `b2`, sorted by the keys, then the reference to `b2b`, if not null, and finally all `B1`s contained in `a1`.
Alternatively, the order can always be overridden by host language means, see Figure 8.

Partiality Any umod visitor can always be applied to any Java object by the method `visit(Object o)`, which by default does nothing. All activities are controlled by the type (= Java class) of the visited model component. In particular, visitor source text is robust against extensions and re-organisations of the model definition.

First-class means that "they can be named, can be passed as arguments, etc." Being Java classes and objects, Visitors are first-class in this sense.

[2] The complete, runnable source code of these examples will be available at `http://bandm.eu`.

```
class Guarded extends MyVisitor {
  @Override public void visit(B2 b) {
    if (b.b2.size()>10)
      super.visit(b);
  }
}
```

Figure 6 Injection of a further guard into some Visitor `MyVisitor` of model `M`.

The classification grid from the section “Rich Variation Points” in [6] is especially useful for identifying the use cases which could be better supported:

Transformation vs. Query Any kind of **query** can be carried out by our basic variant of visitor with *register variables*, see Figure 9.

For transformations there are dedicated variants called `REWRITER` and `COREWRITER`. The former can deal with sharing, internally and between input and output. The latter can deal with cycles transparently. The programmer is responsible for the correct choice, see Figure 10.

Single vs. Cascased Traversal “Cascaded” / “nested” application of visitors is possible by host language means, see Figures 7.

Top-Down vs. Bottom-Up Traversal This is supported by the `MULTIPHASE` visitor which allows independent definitions of the methods `pre` and `post`, which are called before and after descending, resp. The generated code calls as default the visiting methods of the argument’s superclass,

Depth-First vs. Breadth-First Rraversal The usual implementation for visitors is recursive descent and thus depth-first search. If required, the code generator could be extended for breadth-first traversal. Emulation with the current umod code is inconvenient.

Left-to-Right Traversal and vice versa See the discussion of “Traversal” above, and Figure 8.

Types vs. General Predicates as Milestones “Milestones” in this context comes from “Adaptive Programming” [6], is used like “guards” in SP, and decides which model elements shall be processed or not. Umod realizes types = Java classes as static and declarative guards; the programmer can add dynamic and imperative guards by calculated values, see Figure 6.

Full vs. Single-Hit vs. Cut-Off Traversal Figures 11 and 12 show two ways for preemptive termination of a traversal. The first uses the exception mechanism of the host language, while the second completes all pending activities in the normal way, supressing all further descents by a built-in guard variable. The costs of both solutions must be judged by the programmer.

Fixpoint by Equality Test vs. Fixpoint by Failure Applying a umod rewriter means to perform the specified work once. There is no support for automatic fixpoint iteration.

Local Choice vs. Full Backtracking vs. Explicit Cut Umod rewriters support local choice: the commitment to rewritten sub-models can be mixed with branching control flow. Full backtracking is not supported. However, combining local choice with compositionality (sub-rewriters) is fairly expressive.

Traversal With Effects (Accumulation, Cloning, etc.) All effects are under control of the programmer, see the discussions above.

```
class Main extends Visitor {
  public Main(Visitor sub){ this.sub = sub;}
  final Visitor sub ;
  @Override public void visit(B2 b) {
    if (b.b2.size()>10)
      sub.visit(b);
    super.visit(b);
  }
}
```

Figure 7 Visitor calling another visitor.

```
class MyVistor extends Simple {
  @Override public void descend(B2 b) {
    // no call to super.descend(b)
    match(b.b2b); match(b.a); match(b.b2);
  }
  @Override public void descend(B1 b) {
    for (int i = b.b1b.size()-1; i>=0; i-=2)
      match(b.b1b.get(i));
    descend((A)b1);
  }
}
```

Figure 8 Overriding the traversal plan by explicit code.

```
int sum = new Simple() {
  int accu = 0 ;
  public int process(Object a) {
    match(a); return accu;
  }
  @Override public void visit(D d) {
    accu += D.d;
  }
}.process(rootElement);
```

Figure 9 Query realized with a umod visitor.

```
A copy = new Rewriter() {
  final D d17 = new D(17);
  @Override public void visit(D d) {
    replace(d17);
  }
}.rewrite(rootElement);
```

Figure 10 Rewriting realized with a umod visitor.

```
new Visitor {
  class Ex extends RuntimeException{}
  int result = -1;
  public int process(Object a) {
    try { match(a); } catch (EX ex) {}
    return result;
  }
  @Override public void visit(D d) {
    if (d.d > 10) {
      result = d.d;
      throw new Ex();
    }
    super.visit(d);
  }
}.process(rootElement);
```

Figure 11 Terminating a traversal abruptly by an exception.

```
new Visitor2 { // is of MULTIPHASE type
  { hasPre = hasDescend = true; }
  int result = -1;
  public int process(Object a) {
    match(a); return result;
  }
  @Override public void pre(D d) {
    if (d.d > 10) {
      result = d.d;
      hasPre = hasDescend = false;
    }
  }
}.process(rootElement);
```

Figure 12 Terminating a traversal by switching off the descending mechanism.

6 Conclusion

We have demonstrated how the optimization of visitors based on a declarative data model, namely by pruning of unaffected node types according to a static analysis, can be implemented in a purely object-orientied spirit, by transparently subclassing programmer code. The novel implementation technique requires sophisticated tool support for runtime bytecode generation, but results in leaner code and compares well against the naïve baseline and the previous dynamically pruning implementation in empirical benchmarks.

Our approach shows that heterogeneous coding styles and technologies can co-operate in a natural, efficient and well-arranged way, namely declarative model definition and source text generation in the large, imperative coding by the programmer in the middle, and automated low-level bytecode generation in the small. The use of global structural knowledge from the model specification for code optimization purposes appears as a nice supplement and complement to the rather local optimization heuristics of current JIT compilers.

Applying two classification grids from Strategic Programming helps to identify the use cases which are not yet optimally supported by umod.

References

1 Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007. doi:10.1007/978-3-540-73449-9_5.

2 Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C, 2006. URL: http://www.w3.org/TR/2006/REC-xml11-20060816/.

3 Alcino Cunha and Joost Visser. Transformation of structure-shy programs: applied to xpath queries and strategic functions. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 11–20. ACM, 2007. doi:10.1145/1244381.1244385.

4 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

5 Ralf Lämmel. Scrap your boilerplate with xpath-like combinators. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 137–142. ACM, 2007. doi:10.1145/1190216.1190240.

6 Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In William G. Griswold and Mehmet Aksit, editors, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA, March 17-21, 2003*, pages 168–177. ACM, 2003. doi:10.1145/643603.643621.

7 Markus Lepper and Baltasar Trancón y Widemann. Optimization of visitor performance by reflection-based analysis. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations, ICMT 2011*, volume 6707 of *LNCS*, pages 15–30. Springer, 2011. doi:10.1007/978-3-642-21732-6_2.

8 Markus Lepper and Baltasar Trancón y Widemann. Rewriting object models with cycles and nested collections. In *ISoLA 2014, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 445–460. Springer-Verlag, 2014. doi:10.1007/978-3-662-45234-9_31.

9 Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004. doi:10.1145/973097.973102.

10 Ralf Lämmel, Eelco Visser, and Joost Visser. The essence of strategic programming. Unpublished, January 2002. URL: https://www.researchgate.net/publication/277289331_The_Essence_of_Strategic_Programming.

11 Aziz Nanthaamornphong and Rattana Wetprasit. Evaluation of the visitor pattern to promote software design simplicity. *Jurnal Teknologi*, 77(9):61–77, November 2015. doi:10.11113/jt.v77.6186.

12 Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 70–81, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/320384.320391.

13 Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *International Computer Software and Applications Conference (Compsac '98)*. ieee, 1998. doi:10.1109/CMPSAC.1998.716629.

14 Tanumoy Pati and James H. Hill. A survey report of enhancements to the visitor software design pattern. *Software Practice and Experience*, 44(6), 2014. doi:10.1002/spe.2167.

15 Dmitry Petrashko, Ondrej Lhotak, and Martin Odersky. Miniphases: compilation using modular and efficient tree transformations. In *Proc. PLDI 2017*, pages 201–216. ACM, 2017. doi:10.1145/3062341.3062346.

16 Baltasar Trancón y Widemann and Markus Lepper. Improving the performance of the Paisley pattern-matching EDSL by staged combinatorial compilation. In *Declarative Programming and Knowledge Management*, volume 12057 of *LNAI*, pages 268–285. Springer, 2019. doi:10.1007/978-3-030-46714-2.

17 Baltasar Trancón y Widemann and Markus Lepper. LLJava live at the loop – a case for heteroiconic staged meta-programming. In *Proc. MPLR 2021*, pages 113–126, 2021. doi:10.1145/3475738.3480942.

18 Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. In *Proc. FOOL-11*, 2004.

19 W3C. *Scalable Vector Graphics (SVG) 1.1*, 2nd edition, 2011. URL: http://www.w3.org/TR/SVG11/.

20 W3C HTML Working Group. *XHTML 1.0 The Extensible HyperText Markup Language*, 2 edition, 2002. W3C Recommendation. URL: http://www.w3.org/TR/2002/REC-xhtml1-20020801.

Using Spoofax to Support Online Code Navigation

Peter D. Mosses
Delft University of Technology, The Netherlands
Swansea University, UK

Abstract

Spoofax is a language workbench. A Spoofax language specification generally includes name resolution: the analysis of bindings between definitions and references. When browsing code in the specified language using Spoofax, the bindings appear as hyperlinks, supporting precise name-based code navigation. However, Spoofax cannot be used for browsing code in online repositories.

This paper is about a toolchain that uses Spoofax to generate hyperlinked twins of code repositories. These generated artefacts support the same precise code navigation as Spoofax, and can be browsed online. The technique has been prototyped on the CBS (Component-Based Semantics) specification language developed by the PLanCompS project, but could be used on any language after specifying its name resolution in Spoofax.

2012 ACM Subject Classification Information systems → Web interfaces; Software and its engineering → Integrated and visual development environments; Software and its engineering → Development frameworks and environments

Keywords and phrases Spoofax language workbench, name resolution, precise code navigation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.21

Funding The work reported here is part of the PLanCompS project, which was funded by an EPSRC grant to Swansea University from 2011 to 2015 (EP/I032495/1) in collaboration with colleagues at Royal Holloway University of London and City University in London.

Acknowledgements Eelco Visser (1966–2022) was a visionary leader. For more than two decades, I have benefited greatly from reading his persuasive publications, listening to his animated presentations, and discussing topics of common interest with him – especially modular language specification. Since 2012, I have also been an enthusiastic user of the Spoofax language workbench, developed by Eelco and his Programming Languages group at TU Delft. This paper reports on some work that I carried out as a visitor at TU Delft since 2016. I am grateful to the reviewers of a previous version for helpful comments and suggestions.

1 Introduction

Programming languages (including domain-specific languages (DSLs)) usually allow definitions of named entities, and references to entities via their names.

1.1 Code Navigation

In a GitHub Blog post [4], Douglas Creager explains the concept of *code navigation* based on names:

> Code navigation is a family of features that let you explore the relationships in your code and its dependencies at a deep level. The most basic code navigation features are "jump to definition" and "find all references." Both build on the fact that *names* are pervasive in the code that we write.

Although "jump to definition" sounds simple enough, the difficulty of its specification and implementation depends highly on the code language. The determination of the bindings between definitions and references is called name resolution; different languages often have quite different rules for it.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 21; pp. 21:1–21:12
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Name resolution can significantly complicate searching for the definition of a particular name when using an ordinary browser or editor. For example, a language may allow references to a name from other files than the file containing the definition; it may also allow the same name to be defined more than once, possibly with shadowing. For such languages, simple project-wide searches for the definition of a specific name, or for all references to that name, may be imprecise, and return unhelpful false positives.

Eelco Visser led the development of three declarative meta-languages for specifying name resolution: NaBL [11], NaBL2 [19, 25], and Statix [20, 22, 26, 27]. The Spoofax language workbench [8, 10, 31] implements name resolution for any language whose rules are specified in one of these meta-languages. When browsing a file in the specified language, Spoofax equips each reference to a defined name with a hyperlink; clicking on the hyperlink moves the cursor directly to the referenced definition. If the definition is in a different file from the reference, Spoofax automatically opens an editing window on that file. Similarly, Spoofax equips each definition with the list of hyperlinks to all the current references to it. These hyperlinks are precise, and provide reliable support for code navigation.

The languages whose name resolution rules have been specified and implemented in Spoofax include:

- the main Spoofax meta-languages: SDF2, SDF3, Stratego, NaBL, NabL2, Statix, DynSem, Dynamix;
- DSLs used in the production of various software systems: currently WebDSL, IceDust, Green-Marl, PGQL, and LeQuest;
- languages used in Computer Science courses: MiniJava, Jasmin, Tiger, PAPL;
- demonstration languages provided in MetaBorg [14]: SIMPL, QL/QLS, Grace, a subset of Go, MetaC, and Pascal; and
- specification languages developed for other purposes, such as CBS [17] (a meta-language for component-based semantics).

Spoofax users can exploit the hyperlinks between definitions and references to navigate local clones of code repositories in the above languages; but when accessing the same repositories online, those hyperlinks are not available.

1.2 Browsing online code repositories

Suppose that we find a code repository online, and we would like to browse it with precise name-based code navigation using the Spoofax language workbench. Spoofax can only be used to browse local files, so we need to download or clone the repository. We also need to find and download a Spoofax language project for each language used in the code repository (except for the main Spoofax meta-languages). After building those language projects in Spoofax, we can finally browse the cloned repository with precise code navigation.

The necessity of downloading the code repository and the required language projects surely discourages browsing. Moreover, the Spoofax language workbench is currently available only for use in Eclipse and IntelliJ IDEA; users not familiar with either of those IDEs may be reluctant to install them just for browsing some code repository. And users behind company firewalls might not even be allowed to install such 3rd-party software as Spoofax.

Ideally, an online code repository would support precise name-based navigation in standard web browsers, *without* the need to download any files or install new software. The rest of this paper explains how to do that, using Spoofax itself:

- Section 2 gives an overview of a repository that supports precise name-based navigation from references to definitions in web browsers.

- Section 3 presents the Spoofax language project that created the same name-based navigation for local use.
- Section 4 explains how that navigation is made available in web browsers, using a toolchain that combines Spoofax with some standard applications to create "hyperlinked twins" of unlinked code repositories.
- Section 5 relates the Spoofax-based approach to some other approaches that provide name-based code navigation in web browsers.
- Section 6 concludes with suggestions for further development of the presented approach.

2 The CBS-beta Repository

CBS [1] is a meta-language for component-based semantics, developed in the PLanCompS project [21]. See previous publications [3, 17, 28, 29] for motivation, foundations, and explanations of CBS. The CBS-beta repository provides a beta-release of language specifications using CBS. It has two main parts

Funcons-beta: A proposal for an initial library of so-called fundamental programming constructs (funcons).

Languages-beta: Examples of language specifications based on the funcons in Funcons-beta.

(Two further parts are marked as "unstable", and still being developed.)

2.1 CBS specifications

Funcons [18] are *reusable* components: the same funcon can be used, unchanged, in the specifications of many different languages. Funcons correspond closely to concepts of high-level programming languages such as data and control flow, scopes of bindings, mutable variables, streams, abrupt termination, procedural abstraction, etc.

A funcon definition declares the name of the funcon, and specifies its signature: the types of its arguments (if any), and of the value(s) that it returns. The definition also specifies small-step operational semantics rules for evaluating the funcon.

Funcon names are strongly suggestive of the corresponding concepts. The Funcons-beta library defines about 400 funcons. Many of the funcon names are quite long, but have short aliases (e.g., "allocate-initialised-variable" has the alias "alloc-init").

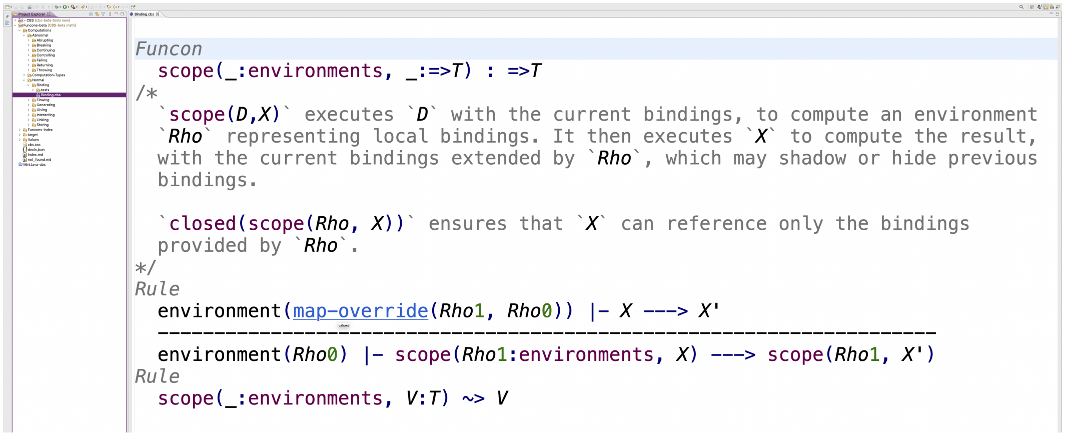
Funcons are to have *fixed* definitions, so *no version control* will be needed for their safe reuse in CBS language specifications. Crucially, adding new funcons does not require any changes to the definitions of existing funcons, thanks to the use of a modular variant of structural operational semantics (MSOS) [15] in CBS.

Apart from illustrating the use of CBS, the aim of the beta-release of CBS and its initial library of funcons was to allow a public review of the definitions, and subsequent adoption of suggestions for improvement, before their finalisation in a full release. The funcon definitions in Funcons-beta have all been validated by empirical testing. The Funcons-beta library has also been found useful in the iCoLa meta-language for incremental (meta-)programming [5].

A language specification in CBS resembles a conventional denotational semantics: it defines the language syntax by a context-free grammar, and it defines semantic functions – mapping phrases in the language to their denotations – by equations. In CBS, the denotations are simply compositions of funcons, instead of higher-order functions.

Languages-beta provides the beta-release of five examples of language specifications in CBS, based on the funcon definitions in Funcons-beta. The examples range from a small introductory example to a substantial sub-language of OCaml.

Table 1 Browsing a local clone of the CBS-beta repository in Spoofax.



```
Funcon
  scope(_:environments, _:=>T) : =>T
/*
  `scope(D,X)` executes `D` with the current bindings, to compute an environment
  `Rho` representing local bindings. It then executes `X` to compute the result,
  with the current bindings extended by `Rho`, which may shadow or hide previous
  bindings.

  `closed(scope(Rho, X))` ensures that `X` can reference only the bindings
  provided by `Rho`.
*/
Rule
  environment(map-override(Rho1, Rho0)) |- X ---> X'
  -----------------------------------------------------------------------
  environment(Rho0) |- scope(Rho1:environments, X) ---> scope(Rho1, X')
Rule
  scope(_:environments, V:T) ~> V
```

Each language specification is independent, and (implicitly) imports all the funcons that it references. Regarding name resolution, the Funcons-beta library corresponds to a single module or package, and users do not need to be aware of the internal file structure of the library.

When browsing funcon definitions and language specifications, precise name-based navigation from references to definitions is essential. Section 3 presents a Spoofax language project for CBS that supports such navigation when using Spoofax to browse a local clone of CBS-beta. Table 1 shows how the definition of the funcon "scope" looks in Spoofax. Apart from the first (defining) occurrence of "scope", all the names in it are references, equipped with hyperlinks to the respective definitions.

2.2 Browsing CBS-beta online

The CBS sources of the specifications in CBS-beta are available in a repository on GitHub. Table 2 shows how the same definition as in Table 1 looks when browsing the CBS sources on GitHub. The text has no hyperlinks, with no support at all for name-based navigation.

However, the CBS-beta repository also contains the sources of an associated website, which includes *hyperlinked twins* of each CBS source file. This website is served by GitHub Pages at `https://plancomps.github.io/CBS-beta/`.

The hyperlinked twins of a CBS source file are in the following formats.

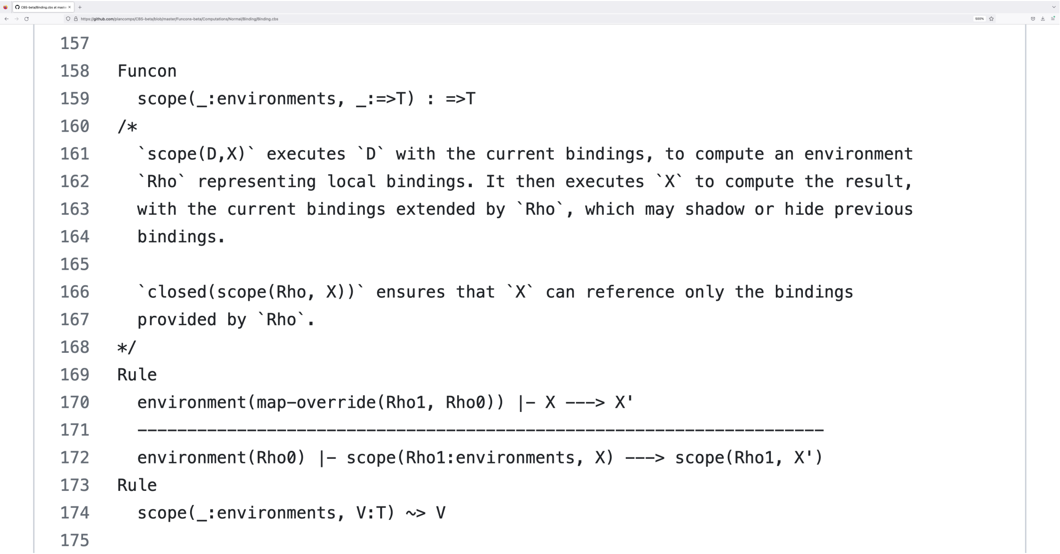
PLAIN: In this format, the web page displays a verbatim copy[1] of the source file. Table 3 shows how the same definition as before looks in the PLAIN format. Names are highlighted: different colours distinguish between names of funcons, syntax sorts, semantic functions, and meta-variables. References are hyperlinked to declarations.

PRETTY: In this format, the web page displays CBS with mathematical typography (as in published articles about CBS). Table 4 shows how the same definition as before looks in the PRETTY format. As in the PLAIN format, names are highlighted, and references are hyperlinked to declarations.

PDF: This format is simply a PDF rendering of the PRETTY web page. Table 5 shows how the same definition as before looks in the PDF format.

[1] The PLAIN format deviates from a verbatim copy by using different font styles.

Table 2 Browsing the CBS-beta repository on GitHub.


CBS-beta/Binding.cbs at maste
https://github.com/plancomps/CBS-beta/blob/master/Funcons-beta/Computations/Normal/Binding/Binding.cbs

```

Funcon
  scope(_:environments, _:=>T) : =>T
/*
  `scope(D,X)` executes `D` with the current bindings, to compute an environment
  `Rho` representing local bindings. It then executes `X` to compute the result,
  with the current bindings extended by `Rho`, which may shadow or hide previous
  bindings.

  `closed(scope(Rho, X))` ensures that `X` can reference only the bindings
  provided by `Rho`.
*/
Rule
  environment(map-override(Rho1, Rho0)) |- X ---> X'
  ---------------------------------------------------------------------
  environment(Rho0) |- scope(Rho1:environments, X) ---> scope(Rho1, X')
Rule
  scope(_:environments, V:T) ~> V

```

The PLAIN and PRETTY hyperlinked twins of a CBS source file include links to each other, to the PDF twin, and to the source file on GitHub.

CBS specifications may include informal text as comments, with embedded references to names. In the source files, comments are enclosed in `/*...*/`, and embedded references in back-ticks; in the hyperlinked twins, the comments are displayed as running text, and the formal CBS specifications are displayed as code blocks.

Section 4 explains how Spoofax is used to generate the hyperlinked twins from the CBS source files.

3 A Spoofax Language Project for CBS

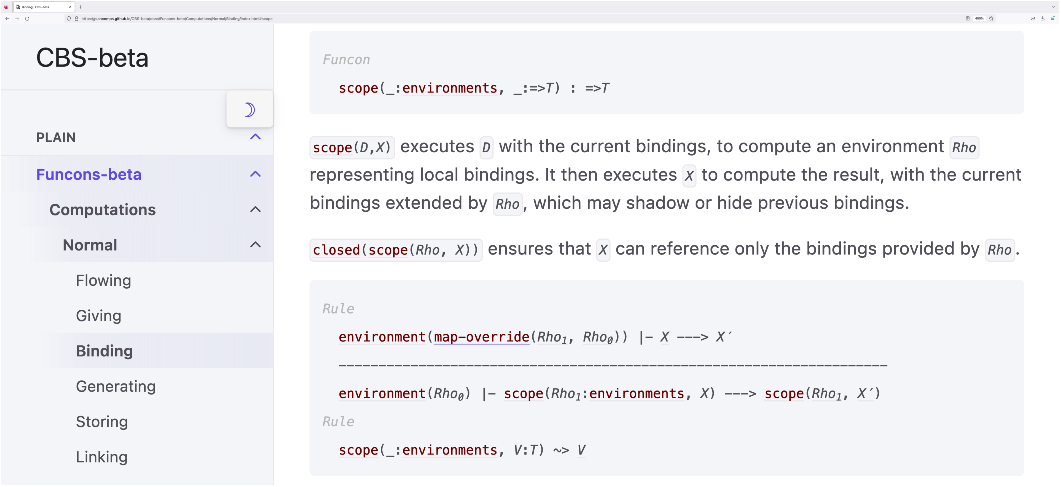
The author has developed a Spoofax language project for CBS. It specifies the syntax of CBS using the meta-language SDF3. Spoofax generates a parser from the SDF3 specification. When a CBS source file is opened, Spoofax creates an editor window for the text, and automatically re-parses the text each time it is edited.

Spoofax uses syntax highlighting to display well-formed phrases; when an edit introduces an error, Spoofax moves the cursor to the source of the error. Spoofax also warns about any phrases that have ambiguous parses.

The syntax of CBS is simpler than that of typical high-level programming languages, and its specification in SDF3 was reasonably straightforward. Unusually, comments are included in ASTs, and formal terms can be embedded in comments (enclosed in back-ticks, as in Markdown).

The CBS language project specifies name resolution using the meta-language NaBL2 [19, 25]. Name resolution in CBS involves checking that all referenced names of each sort (syntax, semantics, funcons) are declared uniquely. Type analysis checks that semantic functions are applied only to the sort of syntax argument specified in their signatures, and that funcons are applied only to the expected number of arguments. However, the analysis of a CBS specification involves the analysis of the entire funcons library, and the implementation of NaBL2 is non-incremental, so re-analysis needs to be suspended while editing even a small CBS source file. (See Section 6 for how to address this drawback.)

Table 3 Browsing a PLAIN hyperlinked twin.



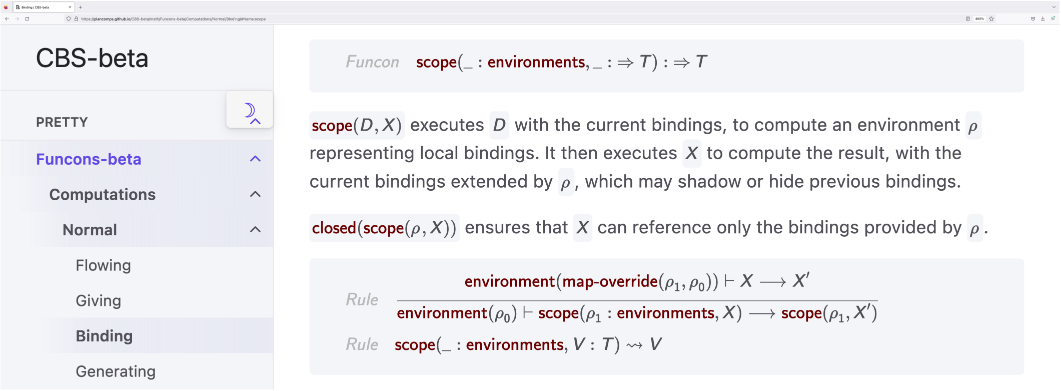
```
Funcon
  scope(_:environments, _:=>T) : =>T
```

`scope(D,X)` executes `D` with the current bindings, to compute an environment `Rho` representing local bindings. It then executes `X` to compute the result, with the current bindings extended by `Rho`, which may shadow or hide previous bindings.

`closed(scope(Rho, X))` ensures that `X` can reference only the bindings provided by `Rho`.

```
Rule
  environment(map-override(Rho1, Rho0)) |- X ---> X′
  ------------------------------------------------------------------
  environment(Rho0) |- scope(Rho1:environments, X) ---> scope(Rho1, X′)
Rule
  scope(_:environments, V:T) ~> V
```

Table 4 Browsing a PRETTY hyperlinked twin.



Funcon $\mathsf{scope}(\_ : \mathsf{environments}, \_ : \Rightarrow T) : \Rightarrow T$

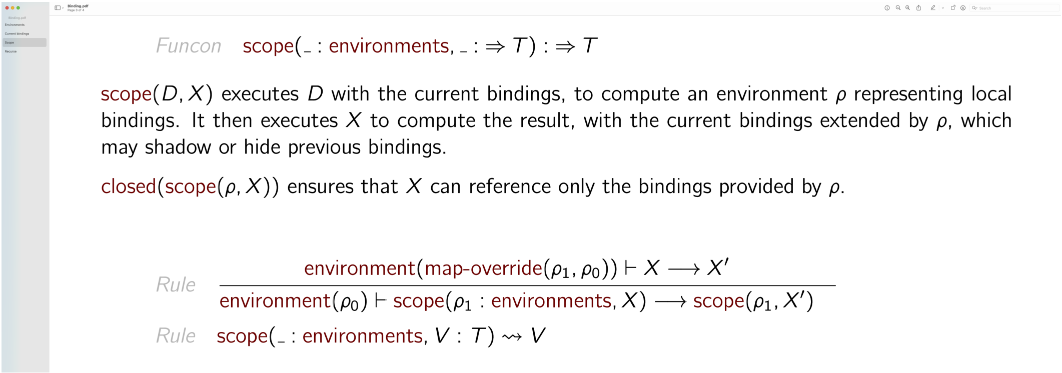
$\mathsf{scope}(D, X)$ executes D with the current bindings, to compute an environment ρ representing local bindings. It then executes X to compute the result, with the current bindings extended by ρ, which may shadow or hide previous bindings.

$\mathsf{closed}(\mathsf{scope}(\rho, X))$ ensures that X can reference only the bindings provided by ρ.

Rule $$\frac{\mathsf{environment}(\mathsf{map\text{-}override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \mathsf{scope}(\rho_1 : \mathsf{environments}, X) \longrightarrow \mathsf{scope}(\rho_1, X')}$$

Rule $\mathsf{scope}(\_ : \mathsf{environments}, V : T) \rightsquigarrow V$

Table 5 Browsing a PDF hyperlinked twin.



Funcon $\mathsf{scope}(\_ : \mathsf{environments}, \_ : \Rightarrow T) : \Rightarrow T$

$\mathsf{scope}(D, X)$ executes D with the current bindings, to compute an environment ρ representing local bindings. It then executes X to compute the result, with the current bindings extended by ρ, which may shadow or hide previous bindings.

$\mathsf{closed}(\mathsf{scope}(\rho, X))$ ensures that X can reference only the bindings provided by ρ.

Rule $$\frac{\mathsf{environment}(\mathsf{map\text{-}override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \mathsf{scope}(\rho_1 : \mathsf{environments}, X) \longrightarrow \mathsf{scope}(\rho_1, X')}$$

Rule $\mathsf{scope}(\_ : \mathsf{environments}, V : T) \rightsquigarrow V$

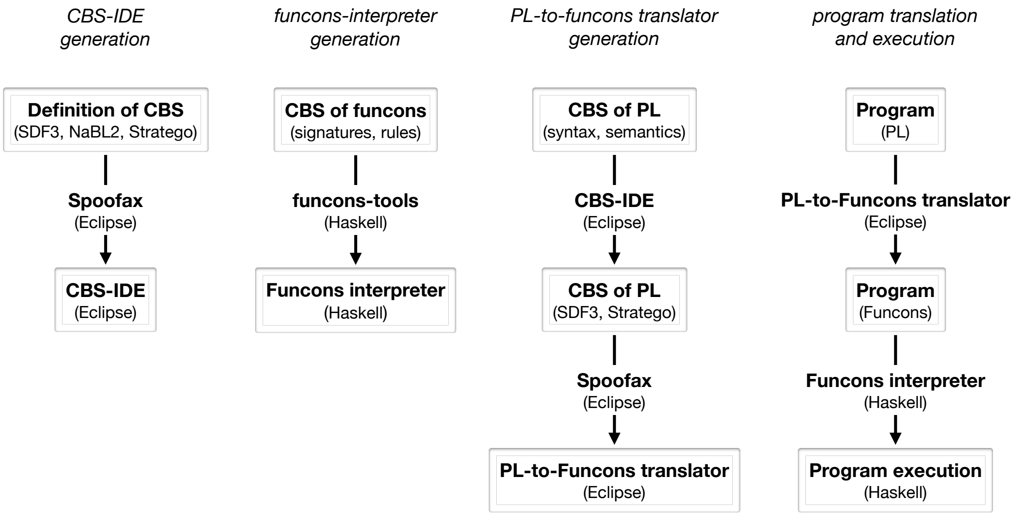

CBS-IDE generation
funcons-interpreter generation
PL-to-funcons translator generation
program translation and execution
Definition of CBS (SDF3, NaBL2, Stratego)
Spoofax (Eclipse)
CBS-IDE (Eclipse)
CBS of funcons (signatures, rules)
funcons-tools (Haskell)
Funcons interpreter (Haskell)
CBS of PL (syntax, semantics)
CBS-IDE (Eclipse)
CBS of PL (SDF3, Stratego)
Spoofax (Eclipse)
PL-to-Funcons translator (Eclipse)
Program (PL)
PL-to-Funcons translator (Eclipse)
Program (Funcons)
Funcons interpreter (Haskell)
Program execution (Haskell)

Figure 1 Generation and use of a CBS-IDE based on Spoofax and external tools [16].

The CBS language project also supports generation of parsers and translators from language specifications. It transforms the context-free grammar of a language specification in CBS to the corresponding SDF3 grammar; it transforms the equations defining the semantic functions to corresponding rewrite rules in Stratego. The resulting language project can parse programs in the specified language, then translate the programs to funcon terms.

External tools (implemented originally in Prolog, and subsequently in Haskell [28]) support evaluation of funcon terms according to the rules that define the funcons, thereby testing whether the rules specify the expected results. Moreover, the combination of these external tools with the Spoofax language project generated from a language specification allows programs in the specified language to be run according to their translation to funcons; this tests the translation rules as well as the rules from the funcon definitions. Figure 1 gives an overview of the tool chain; see reference [16] for further details. *Running the semantics* on suites of test programs, using artefacts generated automatically from the semantics, can reveal subtle errors, as well as eliminating trivial mistakes [9].

4 Generation of Hyperlinked Twins

This section explains how to provide the same name-based code navigation in web browsers as in Spoofax. It uses a toolchain that combines Spoofax with some standard applications to produce *hyperlinked twins* of CBS source files: web pages displaying the same content as the source files, but with the addition of hyperlinks from references to declarations. The web pages also add syntax highlighting, corresponding to that provided by Spoofax when browsing CBS source files locally.

The CBS language project generates hyperlinked twins of CBS source files in three formats: PLAIN, PRETTY, and PDF. For each format, Stratego rules specify a recursive traversal of the analysed AST of the source file, generating strings in the `kramdown` markup language (a variant of Markdown) [12]. The NaBL2 API for Stratego provides strategies to test whether a node of the AST is a declaration or a reference; and when name resolution has determined the declaration to which a reference refers, the API also provides the path of the file that contains the declaration.

Recall that CBS specifications may include informal text as comments. In the source files, comments are enclosed in `/*...*/`; the hyperlinked twins display comments as running text, and `kramdown` determines the layout. Embedded references to names in comments become (highlighted) hyperlinks to declarations. The formal parts of the CBS specifications are displayed as code blocks.

4.1 PLAIN format

The PLAIN format preserves the layout (indentation, line breaks) of the CBS specification by enclosing it in a pre-formatted HTML element (`<pre>`), which is rendered by browsers with a fixed-width font. References and declarations in CBS generate anchor elements (`<a>`) in HTML. Syntax highlighting is produced by HTML span elements. The static site generator Jekyll [6] renders the generated `kramdown` file (prefixed with some meta-data) on the CBS-beta website.

The PLAIN format shows how to write CBS in source files, and the correspondence between a source file and the hyperlinked web page is direct. However, CBS is based on mathematical notation from denotational and operational semantics, and publications about CBS generally display specifications with mathematical typography. The PLAIN format represents an inference rule by a sequence of dashes between the the premises and the conclusion; and it represents a labelled transition by enclosing the label in `--` and `->`. Some of the other approximations of mathematical symbols by ASCII characters are similarly indirect, as well as inelegant. The PRETTY and PDF formats address those issues.

4.2 PRETTY format

For the PRETTY format, the Spoofax language project for CBS generates LaTeX math-mode markup from the analysed ASTs of CBS source files. The `kramdown` variant of Markdown allows such blocks, but leaves their rendering to math typesetting libraries such as KaTeX [7] and MathJax [13].

To avoid dependence on low-level details of the LaTeX commands supported by KaTeX and MathJax, the author has developed CBS-LaTeX [2], a small LaTeX package for CBS specifications. When CBS is marked up using the commands defined by the package, LaTeX formatting produces mathematical typography, suitable for inclusion in published articles. CBS-LaTeX also includes KaTeX and MathJax configurations that produce similar-looking results from the same LaTeX mark-up when embedded in web pages.

Markup using CBS-LaTeX is quite low-level; this makes it easy to adjust the layout to fit the intended page width, but tedious to write. The markup generated by the CBS language project includes line breaks in places where they should enhance readability; it also respects the line breaks in formulae in the source files.

4.3 PDF format

For the PDF format, `kramdown` takes the Markdown with LaTeX math blocks used for the PRETTY format, and converts the Markdown to text-mode LaTeX. Using the CBS-LaTeX package, `pdflatex` then produces a PDF document where the layout and formatting match the rendering of the PRETTY web page in the browser.

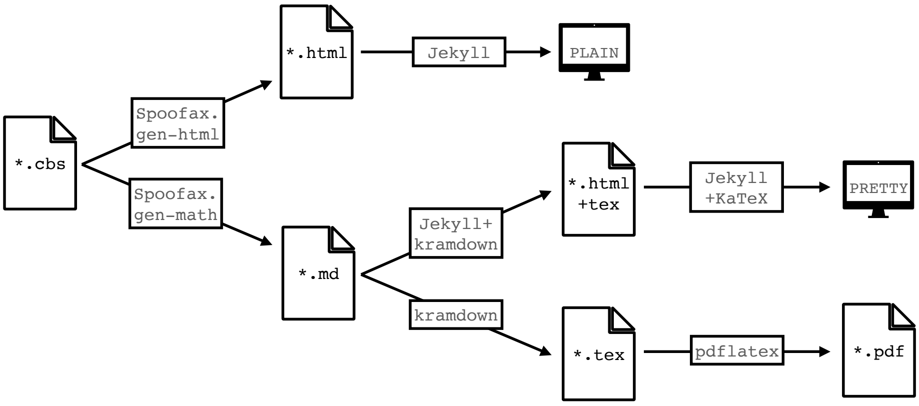


Figure 2 Generation of hyperlinked twins using Spoofax and external tools.

4.4 Offline generation

Although the definitions of funcons are to be fixed (after the finalisation of Funcons-beta) their grouping in files may change. Moreover, the informal comments that motivate and explain the funcons are not definitive, and can change. The Spoofax language project for CBS provides buttons to update the generated Markdown files when needed, and GitHub Pages automatically rebuilds the CBS-beta website when changes are pushed to the CBS-beta repository. However, Spoofax can be used offline: the application Sunshine2 [24] can open Spoofax in Eclipse, build a specific project, and execute the actions attached to buttons. Offline regeneration of all outdated files has been automated using a Makefile.

Figure 2 summarises the toolchains used to generate the PLAIN, PRETTY, and PDF hyperlinked twins of CBS source files.

5 Related Approaches

The generation of websites from Literate Agda specifications provided the initial inspiration for a hyperlinked twin of the CBS source files. In particular, the online book *Programming Language Foundations in Agda* [32] was generated from Literate Agda sources, with name references hyperlinked to definitions. Agda is an indentation-sensitive language, so the generated HTML respects layout in the same way as the PLAIN format on the CBS-beta website. Literate Agda allows informal text to be marked up in Markdown. One difference is that Agda source files make extensive use of Unicode characters, in contrast to the ASCII approximation to mathematical notation used in CBS source files. The implementation of Literate Agda uses `pandoc` to convert Markdown to HTML, analogously to how the CBS-beta website uses `kramdown`.

GitHub currently supports *search-based* code navigation for files in 10 languages: C#, CodeQL, Elixir, Go, Java, JavaScript, PHP, Python, Ruby, and TypeScript. Searching for the definition of a referenced name may return irrelevant results; similarly when searching for all references to a particular definition of a name.

GitHub also supports *precise* code navigation for Python. The name resolution is based on stack-graphs [4], which are closely related to the scope graphs used in Spoofax. Understandably, GitHub is focusing its support for precise code navigation on languages that are widely used in its repositories. The cited reference states:

> Over the coming months, we will add stack graph support for additional languages, allowing us to show precise code navigation results for them as well. Our stack-graphs library is open source and builds on the *Tree-sitter* ecosystem of parsers. We will also be publishing information on how language communities can self-serve stack graph support for their languages, should they wish to.

In principle, it should be possible to migrate the grammar for CBS from SDF3 to *Tree-sitter*, and the name resolution rules for CBS from NaBL2 to stack-graphs. But it appears that doing that would not provide code navigation in CBS repositories on GitHub.

An alternative approach would be to implement CBS support for the Language Server Protocol (which is not currently incorporated in Spoofax). Then web-enabled IDEs (e.g., Visual Studio Code [30]) would be able browse CBS repositories both locally and online. Name resolution for CBS would need to be implemented in a code indexing format such as LSIF or SCIP [23]. That might be straightforward for CBS (due to the uniqueness and global visibility of funcons in Funcons-beta) but perhaps not for the Spoofax meta-languages and the other languages that have already been specified in Spoofax.

6 Conclusion and Future Work

Spoofax has been used in toolchains to generate the PLAIN, PRETTY, and PDF hyperlinked twins of the specifications in the CBS-beta repository. It could be used in the same way for any other repository of specifications in the CBS-beta language, after setting up a Jekyll website in it.

In the CBS-beta repository, each language in Languages-beta is a separate Spoofax language project, with a symbolic link to the Funcons-beta project. On GitHub, other repositories could include Funcons-beta as a submodule, to make the funcon definitions locally available and avoid the need for inter-repository name resolution.

It should be straightforward to generate PLAIN hyperlinked twins of specifications written in the Spoofax meta-languages (SDF3, Statix, etc.), since name resolution for those languages has already been specified in Spoofax. The tree-to-string traversal is specified generically in Stratego for nodes that define or reference names; adding syntax highlighting involves inserting further HTML markup for the relevant nodes, but other nodes generate strings uniformly. To generate PRETTY and PDF twins would require transformation to combinations of Markdown and LaTeX math blocks.

The techniques presented here might be useful for online code navigation also in small programming languages (including DSLs) and specification languages when their syntax and name resolution can be easily specified in Spoofax. For other languages, the alternative techniques to support code navigation discussed in Section 5 may be preferred.

To be able to jump straight to the definition of a funcon from all references to its name is the most important feature of name-based navigation in CBS-beta specifications. Currently, the generated hyperlinked twins do not support finding all references to definitions. The list of all references to a definition is available in the analysed AST, and displayed by Spoofax when browsing the definition; it could be displayed in the same way in the PLAIN and PRETTY twins, but it is unclear how to display it in the PDF twin.

The author is planning to re-specify name resolution for CBS-beta in the Statix meta-language. Statix has an incremental implementation, which will avoid repeatedly re-analysing the Funcons-beta project while editing a language specification. This should remove the main source of inefficiency with interactive use of the current Spoofax language project for CBS-beta, in preparation for a release of the CBS-beta language project as an Eclipse plugin.

References

1 CBS-beta. A framework and meta-language for component-based specification of programming languages, accessed 2023-01-30. URL: `https://plancomps.github.io/CBS-beta/`.

2 CBS-LaTeX. A LaTeX package for CBS specifications, accessed 2023-01-30. URL: `https://plancomps.github.io/cbs-latex/`.

3 Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. *LNCS Trans. Aspect Oriented Softw. Dev.*, 12:132–179, 2015. `doi:10.1007/978-3-662-46734-3_4`.

4 Douglas Creager. Introducing stack graphs. GitHub Blog, 2021. URL: `https://github.blog/2021-12-09-introducing-stack-graphs/`.

5 Damian Frölich and L. Thomas van Binsbergen. iCoLa: A compositional meta-language with support for incremental language development. In Bernd Fischer, Lola Burgueño, and Walter Cazzola, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*, pages 202–215. ACM, 2022. `doi:10.1145/3567512.3567529`.

6 Jekyll. A static site generator, accessed 2023-01-30. URL: `https://jekyllrb.com`.

7 KaTeX. A math typesetting library for the web, accessed 2023-01-30. URL: `https://katex.org/`.

8 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 237–238. ACM, 2010. `doi:10.1145/1869542.1869592`.

9 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012. `doi:10.1145/2103656.2103691`.

10 Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Bootstrapping domain-specific meta-languages in language workbenches. In Bernd Fischer and Ina Schaefer, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 47–58. ACM, 2016. `doi:10.1145/2993236.2993242`.

11 Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. `doi:10.1007/978-3-642-36089-3_18`.

12 kramdown. A library for parsing and converting a superset of Markdown, accessed 2023-01-30. URL: `https://kramdown.gettalong.org`.

13 MathJax. A JavaScript display engine for mathematics, accessed 2023-01-30. URL: `https://mathjax.org/`.

14 MetaBorg. GitHub organisation, accessed 2023-01-30. URL: `https://github.com/metaborg`.

15 Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:195–228, 2004. `doi:10.1016/j.jlap.2004.03.008`.

16 Peter D. Mosses. A component-based formal language workbench. In Rosemary Monahan, Virgile Prevosto, and José Proença, editors, *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*, volume 310 of *EPTCS*, pages 29–34, 2019. `doi:10.4204/EPTCS.310.4`.

17 Peter D. Mosses. Software meta-language engineering and CBS. *J. Comput. Lang.*, 50:39–48, 2019. `doi:10.1016/j.jvlc.2018.11.003`.

18 Peter D. Mosses. Fundamental constructs in programming languages. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*, volume 13036 of *Lecture Notes in Computer Science*, pages 296–321. Springer, 2021. doi:10.1007/978-3-030-89159-6_19.

19 Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

20 Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. Towards language-parametric semantic editor services based on declarative type system specifications. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 26:1–26:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.26.

21 PLanCompS: Programming language components and specifications. Home page, accessed 2023-01-30. URL: https://plancomps.github.io.

22 Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proc. ACM Program. Lang.*, 4(OOPSLA):180:1–180:28, 2020. doi:10.1145/3428248.

23 SCIP Code Intelligence Protocol. Sourcegraph Blog post, accessed 2023-01-30. URL: https://about.sourcegraph.com/blog/announcing-scip.

24 spoofax-sunshine. An application for running Spoofax, accessed 2023-01-30. URL: https://github.com/metaborg/spoofax-sunshine.

25 Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

26 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA):114:1–114:30, 2018. doi:10.1145/3276484.

27 Hendrik van Antwerpen and Eelco Visser. Scope states: Guarding safety of name resolution in parallel type checkers. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark*, volume 194 of *LIPIcs*, pages 1:1–1:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.1.

28 L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. Executable component-based semantics. *J. Log. Algebraic Methods Program.*, 103:184–212, 2019. doi:10.1016/j.jlamp.2018.12.004.

29 L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. Tool support for component-based semantics. In Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki, editors, *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, pages 8–11. ACM, 2016. doi:10.1145/2892664.2893464.

30 Visual Studio Code: Language server extension guide. Visual Studio Code API, accessed 2023-01-30. URL: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide.

31 Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language design with the Spoofax language workbench. *IEEE Softw.*, 31(5):35–43, 2014. doi:10.1109/MS.2014.100.

32 Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. Online book, August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

Towards Modular Compilation Using Higher-Order Effects

Jaro S. Reinders
Delft University of Technology, The Netherlands

Abstract

Compilers transform a human readable source language into machine readable target language. Nanopass compilers simplify this approach by breaking up this transformation into small steps that are more understandable, maintainable, and extensible. We propose a semantics-driven variant of the nanopass compiler architecture exploring the use a effects and handlers to model the intermediate languages and the transformation passes, respectively. Our approach is fully typed and ensures that all cases in the compiler are covered. Additionally, by using an effect system we abstract over the control flow of the intermediate language making the compiler even more flexible. We apply this approach to a minimal compiler from a language with arithmetic and let-bound variables to a string of pretty printed X86 instructions. In the future, we hope to extend this work to compile a larger and more complicated language and we envision a formal verification framework from compilers written in this style.

2012 ACM Subject Classification Theory of computation → Program semantics; Software and its engineering → Compilers

Keywords and phrases algebraic effects and handlers, higher-order effects, monadic semantics, modularity, compilation, nanopass

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.22

Supplementary Material
Software (Source Code): `https://github.com/heft-lang/hefty-compilation`
archived at `swh:1:dir:75f19d9cac0179d90d1a51d1cfe687ea4060cdde`

1 Introduction

The essence of a compiler is a function from a source language, suitable for humans, to a machine language, suitable for computers. As our computers have become more powerful we have seen increasingly complex compilers providing extensive safety guarantees and powerful optimizations. To manage this complexity, modern compilers are designed as a composition of multiple passes. However, the total number of passes has traditionally been kept low for the sake of performance, because each pass adds extra overhead. Thus, compiler passes are more complicated than necessary and therefore harder to *understand*, *maintain*, and *extend*.

To address this problem, Sarkar et al. introduce the nanopass compiler architecture [11]. In the nanopass architecture, each pass is designed to be as small as possible and has only a single purpose. To make development of nanopass compilers easier Sarkar et al. proposed a methodology where each pass only has to specify transformation rules for those language elements which they actually modify. Additionally, intermediate representations of nanopass compilers can be specified by listing language elements which are removed or added to an existing intermediate representation. To address concerns about the performance of this architecture, Keep and Dybvig have developed a competitive commercial compiler using this architecture [6].

In this paper, we present our ongoing work on developing an improved nanopass architecture. As our foundation we use a higher-order effect system [1] which is a state of the art technique for modeling the semantics of programming languages with side effects.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 22; pp. 22:1–22:9
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our approach has the practical advantage of preventing type errors in the compiler and ensuring all cases are covered, which means that the passes defined using our architecture are guaranteed to be syntactically correct by construction. Additionally, our approach abstracts over the control flow giving many of the benefits of continuation-passing-style while retaining a simple monadic interface.

Concretely, we make the following contributions:

- We introduce a novel approach to designing and implementing practical compilers while staying close to formal denotational semantics specifications (Section 3).
- We demonstrate our approach on a simple language with arithmetic and let-bound variables by compiling it to a subset of X86 (Section 3).

2 Monads and Effects

The compiler architecture we propose is based on the concept of monads, algebraic effects and handlers, and higher-order effect trees and their algebras. In this section, we briefly introduce this required background. This section contains no novel work except for the explanations themselves.

2.1 Monads

Monads are a concept from category theory that have been applied to the study of programming language semantics by Moggi [7] and introduced to functional programming by Wadler [17]. A monad as used in this paper consists of four parts:

- a type m
- a binary operator $\gg\!= : m\ a \to (a \to m\ b) \to m\ b$ (known as “bind”)
- a binary operator $\gg : m\ a \to m\ b \to m\ b$
- a function $\mathit{return} : a \to m\ a$

The operator $x \gg y$ is always defined to be equal to $x \gg\!= \lambda z.\, y$ where z is free in y. Furthermore, monads must satisfy three laws:

- left identity: $\mathit{return}\ x \gg\!= \lambda y.\, k\ y = k\ x$
- right identity: $x \gg\!= \lambda y.\, \mathit{return}\ y = x$
- associativity: $x \gg\!= \lambda y.\, (k\ y \gg\!= \lambda z.\, h\ z) = (x \gg\!= \lambda y.\, k\ y) \gg\!= \lambda z.\, h\ z$

We can use monads is to define computations with side effects. An example is the *Maybe* monad which defines partial computations. The type consists of two constructors: $\mathit{Just} : a \to \mathit{Maybe}\ a$ and $\mathit{Nothing} : \mathit{Maybe}\ a$. The bind operator sequences partial computations such that if one subcomputation fails then the whole computation fails. That behavior can be defined using the following two equations: $\mathit{Nothing} \gg\!= \lambda x.\, k\ x = \mathit{Nothing}$ and $\mathit{Just}\ x \gg\!= \lambda y.\, k\ y = k\ x$. The return function is the *Just* constructor. From these definitions it is straightforward to prove that the laws hold, so we will not show the proofs in this paper.

Using the maybe monad we can define a checked division function assuming we have access to an unchecked division function:

$$\mathit{checkedDiv}\ x\ y = \begin{cases} \mathit{Just}\ (\mathit{div}\ x\ y) & \text{if } y \neq 0 \\ \mathit{Nothing} & \text{if } y = 0 \end{cases}$$

Using this checked division function we can define another function that divides a number c by all the numbers contained in a list and return the result in a list:

$$\begin{aligned} &\mathit{divAll}\ c\ \mathit{Nil} &&= \mathit{return}\ \mathit{Nil} \\ &\mathit{divAll}\ c\ (x :: xs) &&= \mathit{checkedDiv}\ c\ x \gg\!= \lambda x'.\, \mathit{divAll}\ c\ xs \gg\!= \lambda xs'.\, \mathit{return}\ (x' :: xs') \end{aligned}$$

In this case, the monad has allowed us to implicitly get the control flow behavior that the whole computation will abort if there is at least one zero in the list.

2.2 Effects

One problem with using monads for specifying the semantics of side effects of programming languages is that you need to define a whole new monad for every programming language, even if many programming languages have side effects in common.

Plotkin and Power addressed this problem by introducing algebraic effects [9], focusing on the effectful operations themselves rather than the concrete monad type. Any set of effectful operations gives rise to a monad for free. In this way, the same operations can be reused in two different modular monads.

In this paper we group effectful operations into units which we call ’effects’. For example we can define the interface of an effect called ’Abort’ as follows:

effect Abort **where**
  $\mathit{abort} : m\ a$

This definition shows that the Abort effect has one operation called ’abort’ which takes no arguments. The type of abort is polymorphic in both the effectful context m and the return type a. In this paper, we assume the obvious implicit constraint that the defining effect, in this case Abort, must be part of the effectful context m.

Any combination of effects gives rise to a monad. So, we can start using the Abort effect to define the *checkedDiv* function instead of using the concrete *Maybe* monad as follows:

$$\mathit{checkedDiv}\ x\ y = \begin{cases} \mathit{return}\ (\mathit{div}\ x\ y) & \text{if } y \neq 0 \\ \mathit{abort} & \text{if } y = 0 \end{cases}$$

The implementation of the *divAll* function does not need to be changed.

Now, let us consider the meaning of the Abort effect. The fact that the *abort* operation claims to produce any polymorphic a as result might seem strange. A normal interface or type class in most programming languages would not be able to implement such a function, because it is impossible to produce a value if the type of that value is not fixed. The reason that we are able to define our effectful operation like this is because one possible side effect is to stop the rest of the computation. That way we do not actually have to produce such a value at all.

In general, to define the meaning of effects we use handlers as introduced by Plotkin and Pretnar [10]. Effect handlers can be thought of as exception handlers with the ability to resume the computation at the location where an effect operation which is handled was used. For the Abort effect, the usual implementation is defined by the following handler:

handle
  $\mathit{abort}\ k \ \ \rightarrow \mathit{Nothing}$
  $\mathit{return}\ x \rightarrow \mathit{Just}\ x$

This example shows that the handler of an effect lists each operation and how it should be handled. In addition to the operations themselves, the handler has access to the continuation k from the point where the operation was used. Furthermore, the Abort handler has a *return* case for when the computation contains no usage of the *abort* operation. For most handlers in this paper, the *return* case is just $\mathit{return}\ x \rightarrow \mathit{return}\ x$ in which case we simply leave it out.

Finally, in this paper we use a higher-order effect system introduced by Bach Poulsen and Van der Rest [1]. Higher-order effect operations are those operations that take effectful computations as arguments. That can be useful for scoping operations, such as exception catching, but also thunks in lazy programming languages. Bach Poulsen and Van der Rest show that it is possible to use their effect system to define interpreters for such higher-order effects. So, it is more than expressive enough for the minimal compiler we present in this paper and even we expect it gives us room to expand our compiler in the future.

3 Compiling with Higher-order Effects

In this section, we present our approach by applying it to a very simple language with arithmetic and let-bound variables. The target language of our compiler is X86 machine code. We explain the required concepts as we develop our compiler for this language. The specifications and compiler passes are presented in a simplified notation, but we have implemented all the work we present here in the Agda programming language [2]. Our code can be found on GitHub[1].

We start off by assuming our parser and possibly type checker has finished and produced an abstract syntax tree which follows the grammar described in Figure 1. Our language has integers, addition, subtraction, negation, a read operation to read an input integer, and a let to bind variables and a var to refer to bound variables.

The abstract syntax is unusual because we reuse the variable binding facilities from our host language in the form of the $v \to \mathit{expr}$ function in the let constructor. This style of abstract syntax is called parametric higher-order abstract syntax (PHOAS) [3]. It allows us to avoid the complexities of variable binding and thus simplify our presentation. We believe other name binding approaches, such as De Bruijn indices [4], could be used instead. However, changing the name binding mechanism would also require changing our representation of effectful computations.

$$\begin{aligned}
\mathit{expr} ::= {} & \mathrm{int}(n) \\
& \mid \mathrm{add}(\mathit{expr}, \mathit{expr}) \\
& \mid \mathrm{sub}(\mathit{expr}, \mathit{expr}) \\
& \mid \mathrm{neg}(\mathit{expr}) \\
& \mid \mathrm{read} \\
& \mid \mathrm{let}(\mathit{expr}, v \to \mathit{expr}) \\
& \mid \mathrm{var}(v)
\end{aligned}$$

Figure 1 Abstract syntax of our simple language with arithmetic and let-bound variables.

The first step of our compilation pipeline will be to denote these syntactic constructs onto an effectful computation. We have chosen to divide our source language into four effects: Int, Arith, Read, and Let.

Figure 2 shows the operations that correspond directly to our source language.

[1] `https://github.com/heft-lang/hefty-compilation`

$$\begin{array}{ll}
\textbf{effect}\ \mathrm{Int}\ \textbf{where} & \textbf{effect}\ \mathrm{Read}\ \textbf{where} \\
\quad \mathit{int} : \mathbb{Z} \to m\ \mathit{val} & \quad \mathit{read} : m\ \mathit{val} \\
 & \\
\textbf{effect}\ \mathrm{Arith}\ \textbf{where} & \textbf{effect}\ \mathrm{Let}\ \textbf{where} \\
\quad \mathit{add} : \mathit{val} \to \mathit{val} \to m\ \mathit{val} & \quad \mathit{let} : m\ \mathit{val} \to (\mathit{val} \to m\ \mathit{val}) \to m\ \mathit{val} \\
\quad \mathit{sub} : \mathit{val} \to \mathit{val} \to m\ \mathit{val} & \\
\quad \mathit{neg} : \mathit{val} \qquad\quad\ \to m\ \mathit{val} &
\end{array}$$

Figure 2 The effects of our source language and their operations with type signatures.

To keep our example simple, we have chosen to use this single type for all values, but we keep the type abstract and simply call it "val". Also note that we now need a special *int* operation to inject integers into this abstract value type.

The reason for keeping the value type abstract is twofold. Firstly, this prevents us from accidentally attempting to use information from these run-time values at compile-time. Secondly, at the end of the compilation, these values stand for register or memory locations. Keeping the values abstract gives us the freedom to choose the concrete representation later on in the compilation pipeline. We use this freedom in the last handler of this section where we convert the program to a string. There we choose the values to be strings.

The first pass of our compiler pipeline maps our abstract syntax from Figure 1 onto the operations we have defined for our source language from Figure 2. This mapping, called a denotation and written using the $[\![\cdot]\!]$ notation, is a recursive traversal of the abstract syntax tree shown in Figure 3. The result of this mapping is a monadic computation involving the Int, Arith, Read, and Let effects.

$$\begin{aligned}
[\![\mathrm{int}(n)]\!] &= \mathit{int}\ n \\
[\![\mathrm{add}(e_1, e_2)]\!] &= [\![e_1]\!] \gg\!\!= \lambda x.\, [\![e_2]\!] \gg\!\!= \lambda y.\, \mathit{add}\ x\ y \\
[\![\mathrm{sub}(e_1, e_2)]\!] &= [\![e_1]\!] \gg\!\!= \lambda x.\, [\![e_2]\!] \gg\!\!= \lambda y.\, \mathit{sub}\ x\ y \\
[\![\mathrm{neg}(e)]\!] &= [\![e]\!] \gg\!\!= \lambda x.\, \mathit{neg}\ x \\
[\![\mathrm{read}]\!] &= \mathit{read} \\
[\![\mathrm{let}(e, f)]\!] &= \mathit{let}\ [\![e]\!]\ (\lambda x.\, [\![f\ \ x]\!]) \\
[\![\mathrm{var}(x)]\!] &= \mathit{return}\ x
\end{aligned}$$

Figure 3 A denotational mapping from our abstract syntax onto our initial set of effectful operations.

Now that we have denoted our syntactic elements into our semantic domain as operations, we can start refining these operations to get closer to the desired target language which is X86 in our case. The effect that we choose to handle first is the Let effect, which only has the *let* operation. We handle this operation by running the right hand side of the binding, passing the resulting value to the body, and finally passing the result of that to the continuation. In code that looks as follows:

$$\textbf{handle}\ (\mathit{let}\ e\ f)\ k \to e \gg\!\!= \lambda x.\, f\ x \gg\!\!= \lambda z.\, k\ z$$

Note that this defines a strict semantics for our let bindings. By using a different handler we could give different semantics to our language. This is an example of the flexibility of algebraic effects and handlers.

$$\begin{array}{l}
\textbf{effect}\ \text{X86}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{addq} & : \mathit{val} \to \mathit{val} & \to m\ () \\
\mathit{subq} & : \mathit{val} \to \mathit{val} & \to m\ () \\
\mathit{negq} & : \mathit{val} & \to m\ () \\
\mathit{movq} & : \mathit{val} \to \mathit{val} & \to m\ () \\
\mathit{callq} & : \mathit{lab} & \to m\ () \\
\mathit{reg} & : \text{Register} & \to m\ \mathit{val} \\
\mathit{deref} & : \text{Register} \to \mathbb{Z} & \to m\ \mathit{val}
\end{array}
\end{array}
\qquad
\begin{array}{l}
\textbf{effect}\ \text{X86Var}\ \textbf{where} \\
\quad \mathit{x86var} : m\ \mathit{val}
\end{array}$$

Figure 4 The effects related to X86 and their operations with type signatures.

At this point, since our language is so simple we can already begin translating into our target language. In Figure 4 we show a minimal subset of X86 that we need to compile our Arith and Read effects. This subset contains in-place arithmetic instructions, the ubiquitous move instruction, the call instruction, and an operation to inject concrete registers into our abstract value type. Additionally, we add an operation to generate fresh variables and inject them into our abstract value type.

We can translate our Arith effect operations into X86 operations by creating a fresh X86 variable, populating it, and then applying the in-place arithmetic operation to the variable. So, we write handler as follows:

$$\begin{array}{l}
\textbf{handle} \\
\quad \begin{array}{lll}
(\mathit{add}\ x\ y) & k \to \mathit{x86var} \gg\!= \lambda z.\, \mathit{movq}\ x\ z \gg \mathit{addq}\ y\ z \gg k\ z \\
(\mathit{sub}\ x\ y) & k \to \mathit{x86var} \gg\!= \lambda z.\, \mathit{movq}\ x\ z \gg \mathit{subq}\ y\ z \gg k\ z \\
(\mathit{neg}\ x) & k \to \mathit{x86var} \gg\!= \lambda z.\, \mathit{movq}\ x\ z \gg \mathit{negq}\ z \gg k\ z
\end{array}
\end{array}$$

The *read* operation requires us to call a function that we will assume is defined in a standard library called read_int. This function places its output in the %rax register, so we have to move it to avoid it being overwritten by other parts of our program. The full definition of our handler for the Read effect is as follows:

$$\textbf{handle}\ \mathit{read}\ k \to \mathit{x86var} \gg\!= \lambda z.\, \mathit{callq}\ \text{read\_int} \gg \mathit{reg}\ \%\text{rax} \gg\!= \lambda x.\, \mathit{movq}\ x\ z \gg k\ z$$

The final challenge to complete this minimal compiler pipeline is to allocate the X86 variables on the stack. Conceptually, this requires us to give each *x86var* operation and give each its own location of the stack. Keeping track of such information in our handler, however, is something we have not yet needed to do for the passes up to this point. Until now, we have handled each effect by translating into other effects directly. Instead, we can parameterize our handlers which means we pass along an extra parameter while handling our operations. Parameterized handlers take one extra parameter and need to pass one extra argument to the continuation[2]. Now we can write the parameterized handler for the X86Var effect which assigns each variable to its own stack location as follows:

$$\textbf{handle}\ \mathit{x86var}\ k\ n \to \mathit{deref}\ \%\text{rbp}\ (-8 \cdot n) \gg\!= \lambda z.\, k\ z\ (n+1)$$

Note that we assume sufficient space is allocated on the stack. Additionally, when applying this handler we need to provide the starting value of the parameter n, which we will choose to be 1.

[2] We ignore effectful subcomputations, because they were already removed in an earlier pass.

At this point, we have a full compiler pipeline from our source language to a subset of X86, but is still in the form of an effectful computation. To get a concrete representation, we implement two handlers for the remaining Int and X86 effects to produce an output string. As part of choosing this concrete representation, we also choose the concrete type for the variables *val* and *lab* to be the string type. We define the handler that turns our effectful computation of Int and X86 effects into a concrete string representation as follows:

$$\begin{array}{lll}
\textbf{handle} & & \\
\quad (\mathit{int}\ n) & k \to & k\ (\mathrm{showInt}\ n) \\
\quad (\mathit{addq}\ x\ y) & k \to & \texttt{"addq "} \mathbin{+\!\!+} x \mathbin{+\!\!+} \texttt{", "} \mathbin{+\!\!+} y \mathbin{+\!\!+} \texttt{"\textbackslash n"} \mathbin{+\!\!+} k\ () \\
\quad (\mathit{subq}\ x\ y) & k \to & \texttt{"subq "} \mathbin{+\!\!+} x \mathbin{+\!\!+} \texttt{", "} \mathbin{+\!\!+} y \mathbin{+\!\!+} \texttt{"\textbackslash n"} \mathbin{+\!\!+} k\ () \\
\quad (\mathit{negq}\ x) & k \to & \texttt{"negq "} \mathbin{+\!\!+} x \mathbin{+\!\!+} \texttt{"\textbackslash n"} \mathbin{+\!\!+} k\ () \\
\quad (\mathit{movq}\ x\ y) & k \to & \texttt{"movq "} \mathbin{+\!\!+} x \mathbin{+\!\!+} \texttt{", "} \mathbin{+\!\!+} y \mathbin{+\!\!+} \texttt{"\textbackslash n"} \mathbin{+\!\!+} k\ () \\
\quad (\mathit{callq}\ l) & k \to & \texttt{"callq "} \mathbin{+\!\!+} l \mathbin{+\!\!+} \texttt{"\textbackslash n"} \mathbin{+\!\!+} k\ () \\
\quad (\mathit{reg}\ r) & k \to & k\ (\mathrm{showReg}\ r) \\
\quad (\mathit{deref}\ r\ n) & k \to & k\ (\mathrm{showInt}\ n \mathbin{+\!\!+} \texttt{"("} \mathbin{+\!\!+} \mathrm{showReg}\ r \mathbin{+\!\!+} \texttt{")"}) \\
\quad \mathit{return}\ x & \to & x
\end{array}$$

4 Related Work

Our work is influenced by Eelco Visser's work on the Spoofax Language Workbench [5]. Eelco Visser was the original designer of the Stratego program transformation language [16] which is part of Spoofax. Stratego can be used to implement a whole compiler back end, however Spoofax was still lacking a way to specifying programming language semantics at a higher level of abstraction.

One step in that direction by Vlad Vergu, Pierre Neron, and Eelco Visser was the DynSem DSL for dynamic semantics specification [14]. Later, Vlad Vergu and Eelco Visser developed a way to improve the performance of programs written in languages specified in DynSem by using just-in-time compilation [15]. However, they were not able to fully eliminate the interpretation overhead that DynSem imposes. Furthermore, DynSem uses big-step operational reduction rules, which require explicit managing of the control flow. For languages which include complicated control flow constructs, such as exceptions, this requires non-trivial glue code in every reduction rule.

In the meantime, Eelco Visser started working on adding type systems to Spoofax. Firstly, together with Van Antwerpen et al., he developed Statix [13], which is a language for defining the static semantics of languages defined in Spoofax. Later, Eelco Visser and Jeff Smits added a gradual type system to the meta-language Stratego [12].

Recently, Thijs Molendijk, who was supervised by Eelco Visser, has developed the Dynamix [8] dynamic semantics specification language for Spoofax. Dynamix is more amenable to compilation and it allows for specifying the semantics of complicated control flow constructs independently from other language constructs. The monadic style of Dynamix makes it similar to our work, but the main difference is that Dynamix has a fixed intermediate representation where the only way to extend it is to add new primitives.

As mentioned in the introduction, our approach embraces the nanopass architecture [11, 6]. The main idea of nanopass compilers is that they consist of many small single purpose passes, which aids understanding. We improve upon this work by making it fully typed to prevent common errors and even check that all cases are covered, and by abstracting over the control flow in the compiler.

5 Conclusions and Future Work

We have presented a new semantics-driven approach to writing compilers by using effect operations as an intermediate representation. We use effect handlers to iteratively refine operations in terms of increasingly lower level operations to finally reach a target machine language.

We have shown a concrete example of this approach applied to a very simple language with arithmetic and let-bound variables. This example application consists of an implementation of a denotation function and handlers which compile this language is compiled in several passes to X86 machine language.

In the future, we would like to extend this minimal compiler with more complicated language constructs such as conditionals, exceptions, and anonymous functions.

Additionally, we would like to implement more complicated analyses on this effectful representation, such as register allocation. We expect these analyses to consist of two stages: first derive concrete structures such as control-flow graphs and interference graphs from our effectful representation, and then perform a pass over the effectful computation that uses the results of the analysis over these structures to transform the program. The first stage would be similar to our handler that turns the effectful computation into a concrete string and the second stage would employ a parameterized handler similar to our stack allocation handler.

Furthermore, we would like to explore the verification of our compilers using algebraic laws for our effect operations, inspired by Interaction Trees [18]. To be specific, we can define a set of laws that describe the behavior of each of the effects in our compiler pipeline. If these laws are sound and complete, with respect to for example definitional interpreters for the effects, then we can prove compiler correctness by proving that these laws are preserved by each of our handlers.

References

1 Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. `doi:10.1145/3571255`.

2 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

3 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, September 2008. `doi:10.1145/1411203.1411226`.

4 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. `doi:10.1016/1385-7258(72)90034-0`.

5 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1869459.1869497`.

6 Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. *SIGPLAN Not.*, 48(9):343–350, September 2013. `doi:10.1145/2544174.2500618`.

7 E. Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989. `doi:10.1109/LICS.1989.39155`.

8 Thijs Molendijk. Dynamix: A domain-specific language for dynamic semantics. Master's thesis, Delft University of Technology, 2022. URL: `http://resolver.tudelft.nl/uuid:8653ab24-a782-41f0-aefc-6b1c8d9a37d5`.

9 Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

10 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

11 Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 201–212, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1016850.1016878.

12 Jeff Smits and Eelco Visser. Gradually typing strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, pages 1–15, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426425.3426928.

13 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

14 Vlad Vergu, Pierre Neron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 365–378, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.RTA.2015.365.

15 Vlad Vergu and Eelco Visser. Specializing a meta-interpreter: Jit compilation of dynsem specifications on the graal vm. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3237009.3237018.

16 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, pages 357–361, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

17 Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/91556.91592.

18 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371119.

Analysing the SML97 Definition: Lexicalisation

Elizabeth Scott
Department of Computer Science, Royal Holloway, University of London, UK

Adrian Johnstone
Department of Computer Science, Royal Holloway, University of London, UK

Abstract

The specification of the syntax and semantics for Standard ML have been designed to support the generation of a compiler front end, but actual implementations have required significant modification to the specification. Since the specification was written there have been major advances in the development of language analysis systems that can handle general syntax specifications. We are revisiting the SML specification to consider to what extent, using modern tooling, it can be implemented exactly as originally written. In this short paper we focus on the lexical specification.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases SML, language specification, lexicalisation, parsing, ambiguity

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.23

1 Introduction

In the Definition of Standard ML [8], the syntax and semantics have been formally specified in a way that is designed to support the generation of a compiler. The lexical syntax is specified in prose and defines a set of lexical tokens. The phrase level syntax is specified using a context free grammar, annotated with some comments, whose terminals are these lexical tokens. We refer to this language as full SML and to the corresponding grammar as the fullSML grammar. The 1997 Definition also includes another grammar, bareSML, specifying a smaller language which is almost, but not quite, a subset of full SML. There is a map from full SML to bare SML defined as a set of rewrite rules which are expressed in terms of the structure given by the fullSML grammar. In fact the rules do not always result in a string in full SML, but this is a technical detail that is not important here. Semantics are defined for bare SML in natural semantics style using SOS inference rules written in terms of the bareSML grammar rules.

The salient point here is that we would like to be able to use the 1997 Definition *as written* as the specification from which a tool that executes semantic evaluation of SML can be directly generated. However, the specification is not in a form that can be directly implemented using classical deterministic approaches. For example, fullSML is ambiguous and some of the specification is in prose or is qualified by textual comments. Actual implementations have required significant, albeit often well documented [9], modification to the structure in the Definition.

Since the Definition was written there have been major advances in the development of systems that can handle general syntax specifications. There are practical general parsers, and our new multi-lexer approach [11] allows the removal of the fixed separation of lexical and phrase level analysis. Our goal is not to review or investigate the modifications required to implement an efficient SML translator; this has been done extensively and the interested reader is encouraged to read Rossberg's discussions [9] related to the development of HaMLet [10] and Kahrs' original analysis [7]. Rather we are revisiting the SML Definition to consider to what extent modern tooling allows the syntax specification to be implemented exactly as originally written. In this paper we focus on the lexical specification.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 23; pp. 23:1–23:12
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The traditional separation of lexical and phrase level specification and analysis, as described for example in [1], significantly restricts the lexical specification. An input string of characters has to be lexicalised to a single token string before parsing, but the lexicalisation is not unique. So the lexer makes disambiguation decisions, typically using heuristics such as longest match and token priority. This is a substantial challenge for the 1997 SML Definition which has several different identifier tokens with the same set of lexemes. One approach is to dispense with the lexer/parser divide and specify the language grammar at character level. This allows the syntax analyser to consider all the lexicalisations, but the corresponding context free grammar is highly ambiguous and creates a substantial disambiguation task. We refer to this approach of using a character level grammar with no supplementary lexical disambiguation techniques as *pure character level parsing*. We have introduced a new form of generalised parser which efficiently parses multiple input strings concurrently [11]. This allows a flexible divide between lexical and syntax level specification, with partial disambiguation carried out at an initial lexical phase. We refer to this as *multi-lexer parsing*. At the same time as the SML Definition was being developed, the need for practical character level GLR parsing in SDF2 was addressed by Visser [14]. Visser's thesis introduced lexical filters which specify disambiguation rules that are applied variously to the underlying character level LR parse table and at parse time. This is often referred to as scannerless parsing, but we will use the term *SGLR parsing* to make the distinction with pure character level parsing clear.

In this paper we discuss some of the SML lexical issues and indicate how they can be addressed using multi-lexer parsing. We also compare multi-lexer SML parsing to pure character level parsing and demonstrate that pure character level parsing is not practical. This shows that the alternatives such as multi-lexer parsing and SGLR parsing are fundamentally required. In future work we intend to compare SGLR parsing with GLL multi-lexer parsing as the two techniques achieve their results in quite different ways.

2 Lexicalisation

In general we follow the terminology in [1]. For a given a set of characters, $\mathcal{A}$, a lexical *token* denotes a set of strings of characters, the token's *pattern*, and a character string in the pattern of a token is called a *lexeme* of that token. A *lexicalisation* of a character string q is a string, $t_0 \ldots t_m$, of tokens with the property that there exist $q_i \in t_i$, $0 \leq i \leq m$, such that $q = q_0 \ldots q_m$.

A *context free grammar* (CFG) consists of a set, $\mathbf{T}$, of terminals, a set, $\mathbf{N}$, of nonterminals disjoint from $\mathbf{T}$, a start symbol, $S \in \mathbf{N}$, and a set of grammar rules $X ::= \alpha_1 \mid \ldots \mid \alpha_t$, one for each nonterminal $X \in \mathbf{N}$, where $\alpha_k \in (\mathbf{T} \cup \mathbf{N})^*$. We refer to this as a BNF grammar. An EBNF grammar is defined in the same way except that the alternates are regular expressions over $\mathbf{T} \cup \mathbf{N}$. It is typical for the terminals of a grammar to be lexical tokens and the terms terminal and token will be used interchangeably.

Classically, the first step in translation is to lexicalise the input character string. We use an end of string character, \$, to avoid having to write special cases for empty strings. For a character string $q\$$ the set $lex(q\$)$ of all lexicalisations over a token set $\mathbf{T}$ can be defined as

$$lex(\$) = \{\$\} \qquad lex(q\$) = \{t_0 t_1 \ldots t_m \in \mathbf{T}^* \mid \exists q_0 (\ q = q_0 p,\ q_0 \in t_0,\ t_1 \ldots t_m \in lex(p\$)\ )\}$$

We can thus assume that lexicalisations are constructed from the left. In effect, for a character string $a_1 \ldots a_d$ we identify lexemes beginning at each index position in increasing order, but we only consider those positions i for which we have already obtained at least one lexicalisation of $a_1 \ldots a_{i-1}$. Then we have a lexical ambiguity at position i if, for some $j > i$,

$a_i \dots a_j$ is the lexeme of two different tokens t, s (intersection ambiguity), or if there is also some $h > j$ such that $a_i \dots a_j$ and $a_i \dots a_h$ are both lexemes (prefix ambiguity). The classical resolution is to specify a priority between t and s, and a longest match condition under which h is chosen over j. We consider both longestWithin, which is only applied to lexemes of the same token, and longestAcross, which is applied even when u and uv are lexemes of different tokens. Of course, priority and longest match strategies do not always allow the required disambiguation to be specified. With a multi-lexer parser [11] more than one lexicalisation can be passed to the parser, allowing the parser to make lexical disambiguation decisions.

2.1 Multi-lexer parsing

The key idea is that the lexer returns token with extent (TWE) elements, (t, i, j), in which t is the token and i, j are the left and right positions of the corresponding lexeme in the input character string. Construction from the left means, for each i, all tokens (t, i, j) are constructed before tokens of the form $(s, i + 1, k)$. Position $i > 0$ is only considered if some TWE element (t, l, i) already exists. Longest match and priority disambiguation are applied at each position i by comparing right extents, and tokens if the right extents are the same.

In a classical lexer one TWE element, (t, i, j), is selected at position i and the lexicalisation continues from position j. In a fully general multi-lexer approach all the TWE elements are constructed and a post-lexer disambiguator may apply disambiguation rules to remove elements. This allows the possibility that more than one lexicalisation can be retained and subsequently parsed. The MGLL [11] generalised LL style parser can efficiently parse multiple lexicalisations of a character string. We do not discuss the technique here but we have used it for the investigations reported below. A third possibility is a hybrid approach in which some lexical disambiguations are applied "on-the-fly" as in the classical lexer, and others are done post-lexing. The difference between this and a full lexer is that, although (t, i, j) may still be constructed, if it is not retained then the lexer may not subsequently consider position j. A hybrid approach is used in SGLR [14], see Section 7. Our current MGLL parser carries out full lexicalisation, although it can easily be modified to run in hybrid form.

Multiple derivations of a given lexicalisation are efficiently represented as shared packed parse forest (SPPF) [3] which is obtained by merging all the corresponding derivation trees. To extend this to represent the derivations of more than one lexicalisation we use lexical extents in the node labels. The leaves of the tree are labelled with TWE elements, (t, i, j), and an interior node is labelled (A, i, j) where i is the left extent of it left-most child and j is the right extent of its right-most child. These trees are then combined by sharing nodes with the same label, and allowing for different sets of children by creating "packed" nodes as children of (A, i, j) and making each family of children the children of a packed node. Where the SPPF corresponds to just one lexicalisation, phrase level ambiguity corresponds precisely to the existence of nodes with more than one packed node child.

3 Aspects of the fullSML grammar

SML program fragments are structured into "phrase classes" that reflect the intended semantics. The fullSML grammar has a nonterminal for each phrase class and grammar rules that specify the sentences of that phrase class. The grammar we use is the same as that in the 1997 SML Definition [8] except that we have converted EBNF optional constructs to BNF and sequences with ellipses, e.g. $(exp_1, \dots, exp_n)$, to recursively defined constructs. For discussion purposes below, part of our grammar is given in Figure 1.

$\mathit{program} ::= \mathit{topdec}\ ;\ \mid\ \mathit{topdec}\ ;\ \mathit{program}$

$\mathit{topdec} ::= \mathit{strdec}\ \mid\ \mathit{sigdec}\ \mid\ \mathit{fundec}$

$\mathit{strdec} ::= \mathit{dec}\ \mid\ \epsilon\ \mid\ \mathit{strdec}\ \mathit{strdec}\ \mid\ \mathit{strdec}\ ;\ \mathit{strdec}$

$\mathit{dec} ::= \texttt{val}\ \mathit{tyvarseq}\ \mathit{valbind}\ \mid\ \epsilon\ \mid\ \mathit{dec}\ \mathit{dec}\ \mid\ \mathit{dec}\ ;\ \mathit{dec}$

$\mathit{valbind} ::= \mathit{pat} = \mathit{exp}$

$\mathit{tyvarseq} ::= \texttt{tyvar}\ \mid\ \epsilon\ \mid\ (\ \mathit{tyList}\ )$

$\mathit{tyList} ::= \texttt{tyvar}\ \mid\ \mathit{tyList}\ ,\ \texttt{tyvar}$

$\mathit{exp} ::= \mathit{infexp}\ \mid\ \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp}\ \mid\ \texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match}\ \mid\ \texttt{fn}\ \mathit{match}$

$\mathit{match} ::= \mathit{mrule}\ \mid\ \mathit{mrule} \mid \mathit{match}$

$\mathit{mrule} ::= \mathit{pat} \Rightarrow \mathit{exp}$

$\mathit{infexp} ::= \mathit{appexp}\ \mid\ \mathit{infexp}\ \texttt{vid}\ \mathit{infexp}$

$\mathit{appexp} ::= \mathit{atexp}\ \mid\ \mathit{appexp}\ \mathit{atexp}$

$\mathit{atexp} ::= \texttt{scon}\ \mid\ \texttt{longvid}\ \mid\ \{\ \}\ \mid\ (\ )\ \mid\ (\ \mathit{exp}\ )$

Figure 1 Part of the fullSML grammar.

The lexical classes are specified in the Definition using textual descriptions. In Figure 2 we give a definition using EBNF which includes set difference, $\setminus$, (only used to remove finite sets), Kleene $()^*$ and positive $()^+$ closure, and ranges, $[0-9]$, $[a-z]$, $[A-Z]$.

$\mathit{vid} ::= \mathit{iden}$

$\mathit{tyvar} ::=\ '\ (\ \mathit{letter} \mid \mathit{digit} \mid '\ )^*$

$\mathit{tycon} ::= \mathit{iden} \setminus \{*\}$

$\mathit{lab} ::= \mathit{iden} \mid [\ 1-9\ ]\ \mathit{digit}^*$

$\mathit{strid} ::= \mathit{alphaNum}$

$\mathit{sigid} ::= \mathit{alphaNum}$

$\mathit{funid} ::= \mathit{alphaNum}$

$\mathit{longvid} ::= (\ \mathit{strid}\ .\ )^*\ \ \mathit{vid}$

$\mathit{longtycon} ::= (\ \mathit{strid}\ .\ )^*\ \ \mathit{tycon}$

$\mathit{longstrid} ::= (\ \mathit{strid}\ .\ )^*\ \ \mathit{strid}$

$\mathit{scon} ::= \mathit{int}\ \mid\ \mathit{word}\ \mid\ \mathit{real}\ \mid\ \mathit{char}\ \mid\ \mathit{string}$

$d ::= \mathit{digit}$

$\mathit{iden} ::= \mathit{idenFull} \setminus \mathit{Rword}$

$\mathit{idenFull} ::= \mathit{letter}\ (\mathit{letter} \mid \mathit{digit} \mid ' \mid \_)^*\ \mid\ \mathit{sym}^+$

$\mathit{alphaNum} ::= \mathit{alphaNumFull} \setminus \mathit{Rword}$

$\mathit{alphaNumFull} ::= \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit} \mid '\ )^*$

$\mathit{sym} ::= \texttt{!} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{\$} \mid \texttt{\#} \mid \texttt{+} \mid \texttt{-} \mid \texttt{:} \mid \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{?} \mid \texttt{@}$
$\qquad\qquad \texttt{\textasciitilde} \mid \texttt{\textbackslash} \mid \texttt{`} \mid \texttt{\textasciicircum} \mid \texttt{|} \mid \texttt{*}$

$\mathit{letter} ::= [a-z]\ \mid\ [A-Z]$

$\mathit{digit} ::= [0-9]$

Figure 2 SML lexical class specification.

3.1 Phrase classes

The phrase class Program is the set of all full SML sentences, and there is a corresponding fullSML start nonterminal *program*. An SML program is a sequence of “top level declarations” whose effects are to modify a top level environment. The nonterminal *topdec* generates the phrase class TopDec whose elements are grouped into subclasses StrDec, SigDec and FunDec (structure level, signature and functor declarations) with corresponding nonterminals *strdec*, *sigdec* and *fundec*. Phrases in StrDec can be concatenated with or without a separating semicolon. Within StrDec we focus on the phrase class Dec, and within Dec we consider the straightforward value declarations, which in turn invoke the expression phrase class Exp. In Figure 1 we have just shown a subset of the constructs, including `if` and `case` statements which are not part of the bare SML language and have to be rewritten using the rewrite rules. The nonterminal *infexp* generates the atomic expressions in the phrase class AtExp of atomic expressions, and also the infix operator syntax. SML allows binding to "patterns" from the phrase class Pat, which are similar in structure to atomic expressions.

As is well documented [9], the nonterminals *strdec* and *dec* are ambiguous: sequencing is not forced to be either left or right associative and sequencing of Decs can be done within Dec or StrDec. These ambiguities, in particular the existence of cycles $strdec \overset{+}{\Rightarrow} strdec$ and $dec \overset{+}{\Rightarrow} dec$, mean that there are infinitely many derivations of any declaration. We have developed an SML-specific cycle removal algorithm which removes specific packed nodes from the SPPF. The remaining ambiguity can be resolved as described in the Definition by choosing alternates by priority. For example choosing $strdec ::= dec$ over $strdec ::= strdec\ strdec$, and "longest match" from the left, so generating as much of the input as possible from the left-most nonterminal in an alternate. We use these disambiguation rules, in the form of SPPF "choosers" see Section 6, to remove phrase level ambiguity, allowing lexical ambiguity to be identified in character level SPPFs[1].

3.2 Lexical classes

The terminals of the fullSML grammar include certain reserved words

```
abstype  and  andalso  as  case  datatype  do  else  end  exception
eqtype fn  fun  functor  handle  if  in  infix  infixr  include  let
local  nonfix  of  op  open  orelse  raise  rec  sharing  sig  signature
struct  structure  then  type  val  with  withtype  where  while
(   )   [   ]   {   }   ,   :   ;   ...   _   |   =>   ->   #   :>
```

We call this set Rword. There is a second small set, Tsym = $\{=,\ *,\ \sim\}$, of terminals that have patterns that are singleton sets containing just themselves, but which are allowed to occur in identifiers. The other terminals have patterns that are specified as lexical classes,

VId TyVar TyCon Lab StrId SigId FunId SCon D LongVid LongTyCon LongStrId

All of these except SCon and D are sets of identifiers. In the first six classes these identifiers

are either sequences of letters, digits, primes and underscores or symbolic identifiers. TyVar identifiers must begin with a prime. The last three classes are extensions that allow identifiers to be linked to modules. The terminal names are lower case versions of the lexical class names.

We do not give the formal definition of the SCon phrase class in this paper, but note that it includes integers, hexadecimals, reals, strings etc in a standard way. The lexical class D, used in the specification that an operator has infix syntax, is just the set of the digits.

[1] The SPPF priority choosers, which are different to the lexical choosers, play a similar role to the SGLR context-free priority specifications [14]. Longest match SPPF choosers allow the disambiguation specification that in the SGLR approach is done using associativity declarations.

3.3 The rewrite rules

The particular syntax specification in the Definition is important. The rewrite rules which convert fullSML to bareSML are associated with the phrase classes, for example expressions in Exp are rewritten to other expressions

`case` *exp* `of` *match* → `(` `fn` *match* `)(` *exp* `)`

We note that the issue of identifier class overlap cannot simply be resolved by merging the classes into a single identifier class and leaving the resolution to post-parse disambiguation. The rewrite rules require the separation of the various identifier lexical classes. For example, the rule

`vid` *atpat* `=` *expr* → `vid` = `fn` $\texttt{vid}_1$ `case` ($\texttt{vid}_1$) `of` (*atpat*) `=>` *expr*

rewrites elements of the derived phrase class FvalBind of function-value bindings. It should rewrite a phrase of the form `vid` (`longvid` : `longtycon`) = `longvid`, but not one of the form `funid` (`strid` : `sigid`) = `longstrid`, which is part of a functor binding phrase class. These phrases would become indistinguishable if the lexical identifier classes were merged.

4 SML lexical disambiguation

With any lexical disambiguation there are two potential problems: (1) a local decision means that ultimately the input cannot be lexicalised or (2) a lexical disambiguation removes all the syntactically correct lexicalisations and the parser rejects the input. The former is not common but perhaps surprisingly the latter does occur, for example longestAcross disambiguation causes the rejection of `x++y` in C-style languages including Java [11]. LongestWithin disambiguation allows a less restrictive approach in which many lexicalisations are removed but some are passed to the parser for resolution.

We consider the lexical classes for SML as actually described in the Definition; so the terminals of the fullSML grammar are the elements of Rword, Tsym and the twelve lexical classes described in Section 3.2.

The Definition states (Section 2.5) that at each lexical stage the longest next item is taken, i.e. longestAcross disambiguation. But this is not always required. The class D and the singleton pattern elements of Rword and Tsym, do not contain lexemes pairs where one is a proper prefix of another, so there is no point in specifying longestWithin disambiguation for these. LongestWithin is specified for the other lexical classes[2].

TyVar does not have any lexemes which belong to or are prefixes of lexemes of any other class, so no further disambiguation specification is required. It is also not necessary to specify longestAcross disambiguation between the other core identifier classes VId, StrId, FunId, SigId, and TyCon. For example, if we have $(\texttt{strid}, i, j)$ and $(\texttt{vid}, i, k)$ where $k > j$, and $q \in$ StrId, $qp \in$ VId are the corresponding lexemes, then $qp \in$ StrId. So there is a TWE element $(\texttt{strid}, i, k)$ and longestWithin on `StrId` removes $(\texttt{strid}, i, j)$. However, it is possible to have TWE elements $(\texttt{vid}, i, j)$ and $(\texttt{longvid}, i, k)$ where $k > j$, $q \in$ VId, $qp \in$ LongVId but $qp \notin$ VId. So we need to specify longestAcross for VId and LongVId.

A full discussion of the degree to which longestAcross is needed is beyond the scope of this paper. In our examples in Section 6 we use full longestAcross, but our multi-lexer parser can handle SML with just longestWithin and indeed with no lexical disambiguation at all.

For TWE sets, post-lexer longestAcross disambiguation can be done at each position i in turn by removing (t, i, j) if there is a (s, i, k) with $k > j$. We can ensure that at least one lexicalisation remains by first pruning the set of TWE elements to ensure that every

[2] LongestWithin for identifier classes can be specified for SGLR parsers using follow restrictions [14], but see the discussion in Section 7.

remaining element belongs to some complete lexicalisation. We refer to this as TWE *dead branch pruning.* LongestAcross disambiguation can also be done during lexicalisation, but this does not easily permit dead branch pruning and we have not done this.

Intersection ambiguity occurs in SML between the nine main identifier classes, and between the classes SCon, Lab and D. The ambiguity cannot be resolved using priority specifications as there is no single class that can be preferred, and the discussion in Section 3.3 shows that we cannot simply merge all the identifiers into a single Id class which is then returned by the lexer.

A partial solution is the so-called Schrödinger's token approach [2] in which the lexer returns a generic *id* token and the parser substitutes the required actual token using phrase level context information. Since the rewriting needs the token string it would be necessary for the parser to rewrite the input token string, and in any case the more general multi-lexer parsing approach can just parse directly all the options represented by a Schrödinger token.

In fact, Section 2.4 of the SML Definition gives a disambiguation of the classes VId, StrId, TyCon and Lab in terms of syntactic context. Classically this would be implemented via some sort of symbol table that shares information between the lexer and other phases of the translation process. The observation that these classes can be disambiguated at the phrase level means that the multi-lexer approach deals with them in a very simple way, all token choices are passed to the parser and the ambiguity is fully resolved as only the "correct" lexicalisation parses. This also applies to the FunId class which is part of the module specification, addressing the FvalBind rewrite issue discussed in Section 3.3.

There is also intersection ambiguity between the identifier classes and their corresponding long classes. This is not covered by the discussion in Section 2.4 of the Definition, and the ambiguity cannot always be resolved by the parser. The problem is illustrated with the phrase $\;$ *if* $x < 1$ *then* y *else* z ;. We expect $<$ to be treated as an infix operator but SML also allows it to be a parameter to an application of a prefix operator x. Thus both lexicalisations below belong to the phrase class Exp.

```
if longvid vid scon then longvid else longvid ;
if longvid longvid scon then longvid else longvid ;
```

In fact is not clear from the Definition whether LongVId is intended to be a lexical or phrase class. It would be possible to turn `longvid` into a fullSML grammar nonterminal and then the lexer will return `vid` rather than `longvid`, avoiding the intersection issue. But this results in a phrase level ambiguity which is equivalent allowing the multi-lexer to return both options, and in either case leaves the parser with an ambiguity to resolve.

We note that formally there is no intersection ambiguity in SML between the keywords and the identifier classes because the 97 Definition does not include keywords in the identifier classes. However, for simplicity of specification our SML identifier lexical classes include the reserved words and the corresponding elements are removed from the TWE set before it is passed to the parser[3].

5 Character level parsing

An alternative approach to handling lexical ambiguity is to define the phrase level grammar at character level. Then lexical classes are singleton sets containing just the corresponding character and lexicalisations are trivial and unique. To facilitate our discussion we shall

[3] This corresponds to the SGLR approach of specifying reject productions [14], which are written in the form `if`→ `VId {reject}`, for keywords such as `if`.

assume that for each lexical class there is a nonterminal in the character level grammar which derives that class. For SML the lexical classes can be specified using regular expressions over characters, and our GLL parser can handle these directly. We create a character level grammar from fullSML by treating the terminals, `vid` etc and the reserved words `case` etc, as nonterminals and adding the rules given in Figure 2. Of course the original lexical ambiguities just become phrase level ambiguities and the problem has been passed to the parser. The parser can construct an SPPF and ambiguities are easily identifiable as nodes that have more than one packed node child. The parser then needs disambiguation rules to decide which of the packed nodes should be retained.

Phrase level ambiguity can be classified as horizontal, two alternates of a nonterminal derive the same string, or vertical, one alternate derives both u and uv [4]. This suggests ambiguity resolution by ordering alternates and applying longest match. This is easy to implement in terms of SPPF packed node removal, and resonates with lexical level priority and longest match disambiguation discussed above. But they are not identical in effect. An interesting question is whether this phrase level disambiguation always gives the same lexicalisation as the one produced using the lexical longest match and priority disambiguation, but this is beyond the scope of this short paper. In fact, the required disambiguation information may not be close in the SPPF to the position of the corresponding multiple packed nodes. For example, there are two syntactically valid derivations of the string `val x = bb`, one in which `bb` is derived from a single *longvid* and one where each `b` is derived from a separate *longvid*. These two derivations are generated from the nonterminal *appexp*, which has two packed node children. We want to choose the packed node that has descendent $(longvid, 8, 10)$ rather than the one that has descendants $(longvid, 8, 9)$ and $(longvid, 9, 10)$, but these nodes are not immediate children of the packed nodes concerned and descendent node searching is required.

In summary, although, using GLL, character level parsing is worst case cubic, applying lexical disambiguation can be difficult, and the SPPFs produced are much larger than those produced using the multi-lexer parser approach, which permits lexical level disambiguation. In the next section we give some data to support this claim.

6 Experimental observations

An ART [6] specification may include EBNF grammar rules, a nominated set of nonterminals called the *paraterminals*[4], disambiguation declarations called *choosers*, term rewrite rules and attribute equations. Lexical choosers specify TWE set disambiguation as discussed in Section 2.1, and SPPF choosers specify syntax level disambiguation via SPPF packed node removal. The set of paraterminals conceptually splits the specification into an upper phrase-level grammar whose terminals are the paraterminals, and one lower lexical grammar per paraterminal, thus specifying a multi-lexer parser. The specification can also be used to generate a pure character level GLL parser, which we call PCLGLL. The advantage of the multi-lexer approach is that lexical disambiguation can be efficiently applied to the TWE set before parsing. In these experiments we use the paraterminal declarations to generate a multi-lexer parser, PTMGLL, which deploys a GLL recogniser for the paraterminal grammars and constructs longest match and priority disambiguated TWE sets.

[4] Paraterminals have a similar role to the SGLR nonterminals A that have an injection of the form `<A-LEX>`$\rightarrow$ `<A-CF>` [14]

The programming artefacts and full experimental results summarised here are available under `art.csle.cs.rhul.ac.uk`. Our corpus comprises the SML source files from the MLWorks compiler and interactive environment originally developed by Harlequin, now open sourced `github.com/Ravenbrook/mlworks`. To avoid the domination of whitespace ambiguity in the character level experiments, the 1,082 files in the flattened source hierarchy were normalised using the `!compressWhitespaceSML` ART directive which replaces all runs of SML whitespace and comments outside of strings with either a single space or a single newline character if the run encompasses a line end. The resulting corpus is `MLWorksSourceFlatCompressed.zip`. The ART specification `smlFull.art` closely follows the Definition, the minor modifications are as mentioned in Section 3, except that we use nonterminals, $wOp ::= w \mid \epsilon$, rather than expansion style replacement, $X ::= \alpha w \beta \mid \alpha\beta$.

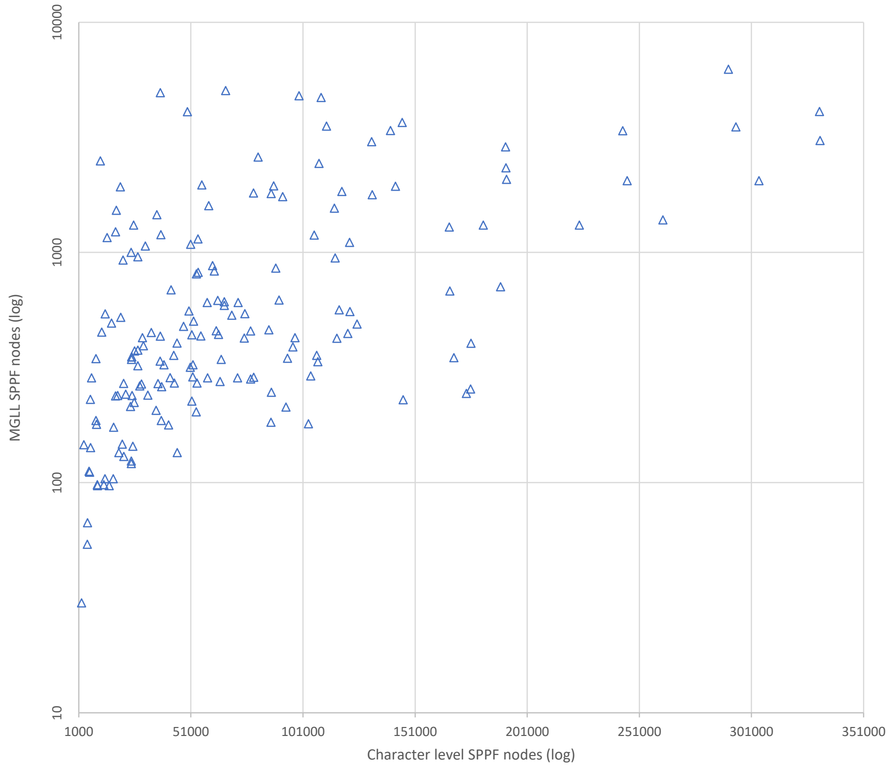


Figure 3 Relative character and paraterminal SPPF size.

Our first observation is about the practicality of a pure character level parser, in terms of the size of the SPPF, which is independent of any particular parsing technique or implementation style. The scatterplot in Figure 3 shows the number of SPPF nodes using log axes: each point represents one file from our corpus. A linear regression analysis gives a trendline with gradient 0.0109, indicating that paraterminal SPPFs are generally around 100 times smaller than pure character level SPPFs. However this broad trend should be treated with caution as is visually evident from the scatterplot: the R^2 value for the linear trendline is only 0.55, and in fact the observed size ratios in this corpus range from 3.74 to 3733.54. The length of each input file is not a useful guide to the size of resulting character level SPPF since SPPF size to input length ratios range from 4.20 to 465.27. So there is no straightforward intuition that allows us to judge the size of a character level SPPF from a piece of SML source. One reason for this is that the SML syntax for function application means that in many cases where a keyword or a single `longvid` is valid, then so is any string of `longvid`s.

In the end, practicality manifests as processing time. Our test system is a DELL XPS 15 9510 laptop with 16GByte of installed memory and an Intel Core i7-11800H eight-core processor running at 2.3GHz under Microsoft Windows 10 Enterprise version 10.0.19042 using Oracle's Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing). On this high end laptop the character level parser can take many seconds to parse some inputs. A limit of 5 seconds of CPU time for each run results in 631 of the 1081 inputs (58%) timing out under PCLGLL: it is clear that using a pure character level grammar is impractical even on a fast modern laptop. None of the files that were processed in less than five seconds by PCLGLL required more than 20ms for PTMGLL to process them.

It is not straightforward to decide how much of the ambiguity in a character level SPPF is lexical. The phrase level SML grammar is ambiguous, so not all the SPPF ambiguity is lexically based, and as shown in Section 3, our individual specifications of each lexical class are unambiguous. Conversely, as discussed in Section 5, a lexical ambiguity can appear in the SPPF under a syntax level nonterminal. ART phrase level choosers in the PTMGLL specification are used to implement the syntax priorities mentioned in Section 3.1. There are 481 strings in the corpus that also only have a single valid lexicalisation (after lexical disambiguation) and we have specified phrase level choosers that fully disambiguate the derivations of these sentences.

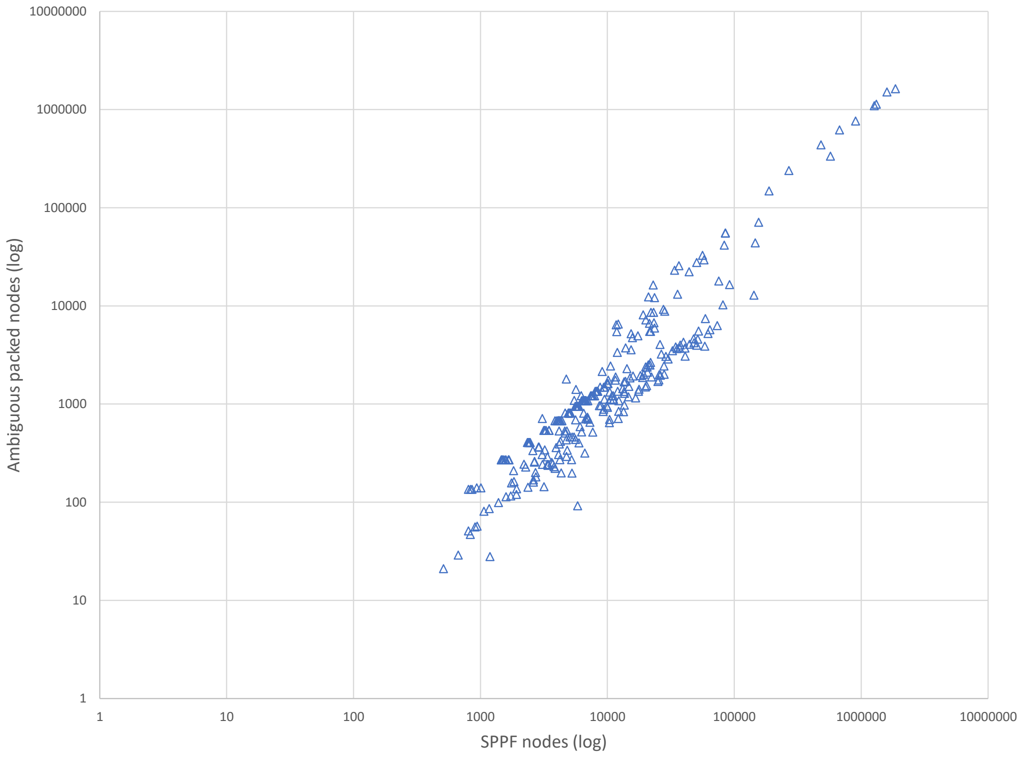


Figure 4 Residual ambiguity.

Of these 481 strings, 165 are not parsed by the PCLGLL parser in within 5 seconds. For the other 316 strings, applying these choosers in the PCLGLL parser removes all the phrase level ambiguity, and the remaining ambiguity is thus lexical. The scatterplot in Figure 4 shows the number of “ambiguous” packed nodes (i.e. packed nodes with siblings) in the character level SPPF plotted against the total number of nodes. There are 51 strings for which the ambiguous packed nodes comprise greater than 20% of the total SPPF nodes, and in three cases the proportion rises to over 90%.

These observations make concrete the commonly held view that pure character level parsing is not viable; hybrid techniques involving some direct lexical disambiguation are needed.

7 Scannerless GLR

At the time of the 1997 SML Definition, tools that could handle general context free specifications were already emerging. ASF+SDF [13] has a GLR [12] based parser, but running this on pure character level grammars in practice triggers the practicality issues illustrated above.

As we have already remarked, Visser addressed the issue in his PhD thesis [14] by adding lexical filters to SDF and developing associated Scannerless GLR parsers. Modifications to SGLR were introduced for use with an RNGLR parser [5] but the approach remains the same. In Visser’s thesis the disambiguation heuristics are expressed in terms of derivation selection, but the implementation in an SGLR parser relies heavily on the LR-parsing technique. For example, follow restrictions, associativity and context-free priorities are implemented as modifications to the LR parse table, so comparison with TWE set disambiguation is not straightforward and merits further investigation.

We conclude with the observation that the SML lexical class SCon contains both real numbers such as 1.2 and “words” such as 0w1. So for lexical longest match SCon requires that a real number should not be followed by a period, but a word can be. Thus an SGLR-style follow restriction is not directly available to implement longest match for SCon, the restrictions need to be considered separately for different SCon subclasses. It is likely implementing the SML 97 Definition (as written) using SGLR will generate interesting challenges that Eelco Visser would have relished.

References

1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools, 2nd edition.* Addison-Wesley, 2006.

2 John Aycock and R. Nigel Horspool. Schrodinger’s token. *Software: practice and experience*, 31:803–814, 2001.

3 Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th conference on Association for Computational Linguistics*, pages 143–151. Association for Computational Linguistics, 1989.

4 Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, March 2010. Earlier version in Proc. 12th International Conference on Implementation and Application of Automata, CIAA ’07, Springer-Verlag LNCS vol. 4783.

5 Giorgios Economopolous, Paul Klint, and Jurgen J. Vinju. Faster scannerless GLR parsering. In *CC’09*, volume 5501 of *Lect. Notes Comput. Sci.*, pages 126–141. Springer, 2009.

6 Adrian Johnstone and Elizabeth Scott. Translator generation using ART. In M.van den Brand B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 306–315. Springer-Verlag, 2011.

7 Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical report, University of Edinburgh, Technical Report ECS-LFCS-93-257, 1994.

8 Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

9 Andreas Rossberg. Defects in the revised definition of Standard ML. `https://people.mpi-sws.org/~rossberg/papers/sml-defects-2013-09-18.pdf`, 2013.

10 Andreas Rossberg. Hamlet; SML reference interpreter. `https://people.mpi-sws.org/~rossberg/hamlet/`, 2013.

11 Elizabeth Scott and Adrian Johnstone. Multiple lexicalisation - A Java based case study. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE'19*. ACM, 2019.

12 Masaru Tomita. *Generalized LR parsing*. Kluwer Academic Publishers, 1991.

13 M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

14 Eelco Visser. Scannerless generalised-LR parsing. Technical Report P9707, University of Amsterdam, 1997.

A Simply Numbered Lambda Calculus

Friedrich Steimann
Fernuniversität in Hagen, Germany

Abstract

While programming languages traditionally lean towards functions, query languages are often relational in character. Taking the *relations language* of Harkes and Visser as a starting point, I explore how the functional paradigm, represented by the lambda calculus, can be extended to form the basis of a relational language. It turns out that a straightforward extension with strings of terms not only supports surprisingly many features of the relations language, but also opens it up for higher-order relations, one prominent feature the relations language does not offer.

2012 ACM Subject Classification Software and its engineering → Multiparadigm languages; Theory of computation → Functional constructs

Keywords and phrases multiplicities, strings, lambda calculus, relational programming

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.24

Related Version *Full Version*: `https://feu.de/ps/pubs/SNLC-TR.pdf` [7]

Acknowledgements I thank Sebastian Erdweg for the joint exploration of a lambda calculus with plural terms on the last day of Dagstuhl Research Meeting 21183. My anonymous reviewers greatly helped improve this paper.

I am forever grateful for the friendliness and collegiality of Eelco Visser that I enjoyed.

1 Introduction

At 2014's instalment of the Software Language Engineering (SLE) conference[1], Eelco Visser and his then PhD student Daco Harkes presented their *relations language* [1], a language designed for querying the object graphs that represent the data of various web-based information systems [2]. The language features first-class, n-ary, bidirectional relations between attributed objects and offers concise path expressions (using the usual dot notation; e.g., `x.children.age`) for convenient querying and navigation. To harden the language with static guarantees, the places of relations are constrained not only by types, but also by so-called *native multiplicities*, which abstract from the number of objects other objects may relate to in each place, and which are said to be *orthogonal to types* [1, 2].

To grant some generality in query expressions, the relations language defines an arithmetic sublanguage whose operators accept arbitrarily many operands in each place. For instance, addition can handle multiple numbers in the places of both summands, and even no numbers (in other languages represented by a null value). This is expressed by (big-step) evaluation rules of the kind

$$[\textsc{Add}]\ \frac{e_1 \Downarrow V_1 \qquad e_2 \Downarrow V_2}{e_1 \oplus e_2 \Downarrow \{\!|\, v_1 + v_2 \mid v_1 \in V_1, v_2 \in V_2 \,|\!\}}\,,$$

in which V_1 and V_2 are (flat) bags (here delimited by braces $\{\!|$ and $|\!\}$) and which means that if e_1 evaluates to $\{\!|\, 1, 2 \,|\!\}$ and e_2 to $\{\!|\, 3, 4 \,|\!\}$, then $e_1 \oplus e_2$ evaluates to $\{\!|\, 4, 5, 5, 6 \,|\!\}$. Multiple numbers naturally result from accessing numeric attributes (such as `age`) on multiple objects

[1] Starting with its inaugural issue in 2008, Eelco Visser contributed a total of 18 papers to the SLE conference series, and served as its General Chair only last year (2021).

(as they may result from query expressions such as `x.children`) in one expression (e.g., `x.children.age`). Note that since multiplicity is not encoded in type (the two are thought to be orthogonal), the type system of the relations language grants addition of pairs of multiple numbers without further measures (such as coercions). And yet, these kinds of additions appear to be of little use in a query language[2], unless multiplicities are constrained to "zero or one" (multiplicity ? in the relations language), in which case it amounts to adding under the conditions of an `Optional` type [1, 2]. For instance, if e_1 evaluates to $\{\!|\, 1 \,|\!\}$ (corresponding to `Some 1`) and e_2 to $\{\!|\;|\!\}$ (corresponding to `None`), then, by above rule ADD, $e_1 \oplus e_2$ evaluates to $\{\!|\;|\!\}$. Having the semantics of `Optional` for free is certainly a commendable feature of the relations language.

While adding multiple numbers through ADD appears to be of little use in querying, aggregating multiple numbers is certainly central. This is also acknowledged by the relations language, which introduces special aggregation operations for this purpose. Evaluation of these operations is defined through pairs of rules like

$$[\textsc{Sum}]\ \frac{e \Downarrow \{\!|\, v_1, \ldots, v_n \,|\!\} \qquad n \geq 1}{\mathrm{sum}(e) \Downarrow \{\!|\, \Sigma_{i=1}^{n} v_i \,|\!\}} \qquad\qquad [\textsc{Sum0}]\ \frac{e \Downarrow \{\!|\;|\!\}}{\mathrm{sum}(e) \Downarrow \{\!|\, 0 \,|\!\}}\,.$$

Note that while ADD distributes addition over its multiple operands and collects the results, SUM must treat the multiple as a whole.

While the relations language is very general in its coverage of arithmetic (and logical) operations (see, e.g., [6] for a more constrained approach), it is somewhat less so in other respects. Specifically, even though relations are first-class, it does not feature higher-order relations (relations of relations). Like higher-order functions, higher-order relations would not only let users define their own operations, but would also allow the representation of higher-degree (n-ary) relations as nested binary relations (in analogy to the currying of functions), thereby increasing the expressiveness of the language while at the same time reducing its size.

With this tribute to Eelco Visser's work, I aim to provide a common basis for the core constructs of the relations language, navigation of relations and computing with multiple numbers. I do this by extending the lambda calculus (LC) with strings of elementary terms and values. By associating different multiplicities (in this work called *numbers*, alluding to the grammatical category *number* with values *singular* and *plural* [6]) with strings, I not only show that the resulting calculus, which I have pretentiously dubbed "simply *numbered* lambda calculus" (SNLC), has interesting safety properties, but also shed new light on the relation of typing and numbering which, when moving to higher-order functions (or relations), appear to be parallel instead of orthogonal as previously thought: specifically, while number and type may be independent, the structure of the here introduced number specifiers parallels that of function types.

2 Extending the Lambda Calculus with Strings of Terms

Rather than using lists (which can be encoded in the pure LC, but lead to a polymorphic typing discipline) to represent multiples, I will extend the LC with strings. Compared to other monoids [8], strings have the advantage of being purely syntactical (with concatenation as the monoid operation); unlike lists, strings are inherently flat (there are no strings of strings; Section 5 will spell out what this means).

[2] see [8] for a discussion

2.1 Syntax

The syntax of my extended LC has elementary terms, e, and strings of elementary terms, t, which are terms, too:

$$t ::= \overline{e}$$
$$e ::= x \ \mid\ \lambda x\,.\,t \ \mid\ t\ t$$

To spell out a string $\overline{e}$, I write $e_1 \cdot \ldots \cdot e_n$ instead of $e_1 \ldots e_n$; I do so to distinguish the string $e_1 \cdot e_2$ unmistakably from the function application $e_1\ e_2$.[3] Note that in a function application $t_2\ t_1$, (non-elementary) strings can occur in three places: at the argument position (t_1), at the function position (t_2), and at the function body positions.

For a term $t = e_1 \cdot \ldots \cdot e_n$, I say that t has length n. For $n = 0$ (the empty string of terms), I write ϵ, to which I refer as *nothing*; e.g., the function $\lambda x\,.\,\epsilon$ is said to return nothing. For terms $t_1 \cdot \ldots \cdot t_n$ (which have the shape of $\overline{e}$ and hence that of t), I sometimes write

$$\Pi_{i=1}^{n} t_i\ .$$

Note that e also has the shape of t; an elementary term is a string term with length 1 and vice versa.

2.2 Operational Semantics

I define small-step operational semantics of my extended LC using a reduction relation $\longrightarrow$. Using this relation, terms t are reduced to values, which can be elementary (u) or strings of elementary values (v):

$$v ::= \overline{u}$$
$$u ::= \lambda x\,.\,t$$

According to this grammar, ϵ is a (non-elementary) value. Note that the fact that a term is elementary does not mean that it reduces to an elementary value; specifically, $t_2\ t_1$ may reduce to a non-elementary value. Constraining terms so that they are guaranteed to reduce to elementary values is achieved by a *number system*, to be introduced in Section 3.

The rules defining the reduction relation are the following:

$$[\textsc{R-Str}]\ \frac{v \cdot t \neq \epsilon \qquad e \longrightarrow t'}{v \cdot e \cdot t \longrightarrow v \cdot t' \cdot t} \qquad [\textsc{R-App1}]\ \frac{t_2 \longrightarrow t_2'}{t_2\ t_1 \longrightarrow t_2'\ t_1} \qquad [\textsc{R-App2}]\ \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$[\textsc{R-AppS}]\ (\Pi_{i=1}^{n} \lambda x\,.\,t_i)\ (\Pi_{j=1}^{m} u_j) \longrightarrow \Pi_{i=1}^{n} \Pi_{j=1}^{m} [u_j/x] t_i$$

$$[\textsc{R-AppP}]\ (\Pi_{i=1}^{n} \lambda x\,.\,t_i)\ v \longrightarrow \Pi_{i=1}^{n} [v/x] t_i$$

As usual, I write $\xrightarrow{*}$ for the transitive closure of $\longrightarrow$. The following is of note:

- R-Str, R-App1 and R-App2 are congruence rules [3]: R-Str transforms strings having lengths greater than 1 to value strings, while R-App1 and R-App2 transform (elementary) applications to (elementary) applications comprised of value strings. Note that R-App2 means that I rely on call-by-value.

[3] Note that $\cdot$ is not an operator of the object language here; rather, it may be read as the concatenation operator of the metalanguage (and a such is a sibling of substitution).

- For reducing a term $v_2\ v_1$ (the application of a value string to a value string), the choice between the computation rules R-AppS and R-AppP is ambiguous. For instance, $(\lambda x\,.\,x)\ u \longrightarrow u$ both by R-AppS and by R-AppP. This ambiguity will be resolved below (via numbering of terms). Note here that R-AppP substitutes a string for x, whereas R-AppS substitutes each element of the string separately. For instance,
  - using R-AppP, $(\lambda x\,.\,t_1)\cdot(\lambda x\,.\,t_2)\ u_1\cdot u_2 \longrightarrow [u_1\cdot u_2/x]t_1\cdot[u_1\cdot u_2/x]t_2$, while
  - using R-AppS, $(\lambda x\,.\,t_1)\cdot(\lambda x\,.\,t_2)\ u_1\cdot u_2 \longrightarrow [u_1/x]t_1\cdot[u_2/x]t_1\cdot[u_1/x]t_2\cdot[u_2/x]t_2$.

  This difference, which parallels that between the the holistic application of sum and the distributive application of $\oplus$ from Section 1 and which, in the context of nondeterminism, has been described as that between *plural* and *singular semantics* [4, 8], is perhaps most prominent when applying a function to ϵ: under singular semantics (R-AppS), application is strict with respect to ϵ (using R-AppS, $(\lambda x\,.\,t)\ \epsilon \longrightarrow \epsilon$ for every term t), whereas under plural semantics (R-AppP), it is not generally (using R-AppP, $(\lambda x\,.\,t)\ \epsilon \longrightarrow [\epsilon/x]t$).
- As an immediate consequence of the above, using R-AppP, a function application reduces to an elementary value if $n = 1$ (i.e., there is only one function to be applied) and if $[v/x]t_1$ reduces to an elementary value; using R-AppS, the same result additionally requires that $m = 1$, i.e., that the argument of the application is elementary, too. This will be reflected in the numbering of terms as introduced next.

3 Numbering of Terms

Rather than the typing that leads to the simply typed lambda calculus (STLC), I introduce numbering to abstract from the results of computations, and to specify the well-formedness of terms.

3.1 Numbers

Rather than types, I introduce *numbers* η as abstractions of the values terms t reduce to. Number has two possible instances: !, meaning "length 1" (an elementary value; also referred to as "exactly one"), and $*$, meaning "any length" ("any number"). To express that the latter includes the former, I let $! < *$ and define max on numbers η accordingly: $\max(\eta_1, \ldots, \eta_n)$ equals $*$ if $\eta_i = *$ for any $1 \leq i \leq n$, and equals ! otherwise. Numbers are thus *numeric* abstractions: they do not abstract from the kind of a value as types do (e.g., integer or boolean), but from its length. For a term t having number !, one may expect that the value it reduces to has length 1 (i.e., is elementary); for a term having number $*$, one may expect that it reduces to a value of any length[4].

With numbers in mind, one can easily observe that using R-AppS, the variable x is always replaced (via substitution) with a value having number ! (an elementary value), whereas using R-AppP, x is replaced with a value having number $*$ (a value string). Making this explicit by annotating the variables x of functions $\lambda x\,.\,t$ with either

- !, to express that only elementary values will be substituted for x, or with
- $*$, to express that values of arbitrary length may be substituted for x,

disambiguates between the rules R-AppS and R-AppP in the reduction of function application: $\lambda x!\,.\,t\ v$ calls for R-AppS while $\lambda x*\,.\,t\ v$ calls for R-AppP. Tying the disambiguation to the syntax of functions rather than that of applications acknowledges that in the context of multiple arguments (plurals), we may want to distinguish between distributive and holistic

[4] Note how this is more than expecting nothing: one may still expect that it reduces to a value!

treatment, and that this distinction is tied to the function (or operator; e.g., $\oplus$ vs. sum; see Section 1), not the application. Note that the same distinction could be achieved by introducing a second abstraction symbol (such as Λ, complementing λ) for functions and inferring the number of each variable x from the symbol introducing it; however, my choice of using number annotations (which will be needed anyway, as shown next) lets my extension of the LC appear gentler.

3.2 Mapping Constraints and Number Specifiers

It turns out that simple number annotations on variables are not sufficient to infer the number of the value a term will reduce to: Since functions are values that can be substituted for the parameters (variables) of other functions, in whose bodies their application may determine the result of applying the host function, we need to annotate each function (and the variables for which it may be substituted) with the number of its parameter and that of the term constituting its body. For this, I introduce *number specifiers* π defined by the grammar

$$\pi ::= \eta\langle\mu\rangle \qquad\qquad \eta ::= \,! \;\mid\; * \qquad\qquad \mu ::= \Box \;\mid\; \mu \xrightarrow{\eta\,\eta} \mu$$

where I refer to μ as a *mapping constraint* (named after the mapping constraints 1:1 and 1:N from relational database theory). Mapping constraints are defined recursively to account for higher-order functions; $\Box$ terminates the recursion, thereby playing the role of a base type in the STLC. Like base types, $\Box$ demands the existence of values that are not functions, that is, of constants c: the number specifier $*\langle\Box\rangle$ is thus to be read as "any number of constants", $!\langle\Box \xrightarrow{!\,*} \Box\rangle$ as "one function with singular semantics, mapping one constant to any number of constants", and $*\langle\Box \xrightarrow{*\,!} \Box\rangle$ as "any number of functions with plural semantics, each mapping any number of constants to one constant".

Given number specifiers and constants, I extend the syntax of my extended LC further (changes highlighted):

$$\begin{aligned} t &::= \overline{e} \qquad\qquad & e &::= x \;\mid\; \lambda x\pi\,.\,t \;\mid\; t\;t \;\mid\; c \\ v &::= \overline{u} \qquad\qquad & u &::= \lambda x\pi\,.\,t \;\mid\; c \end{aligned}$$

The rules governing reduction of function application are then adapted as follows:

$$[\text{R-A}\textsc{pp}\text{P}]\;\; (\Pi_{i=1}^{n} \lambda x {*}\langle\_\rangle\,.\,t_i)\; v \longrightarrow \Pi_{i=1}^{n} [v/x] t_i$$

$$[\text{R-A}\textsc{pp}\text{S}]\;\; (\Pi_{i=1}^{n} \lambda x !\langle\_\rangle\,.\,t_i)\; (\Pi_{j=1}^{m} u_j) \longrightarrow \Pi_{i=1}^{n} \Pi_{j=1}^{m} [u_j/x] t_i$$

In each rule, the wildcard _ (which is not an element of the object language) in the number specifier $\eta\langle\_\rangle$ of the formal parameter x stands for a mapping constraint that is irrelevant for the (selection of the) rule. The number η on the other hand chooses between singular and plural semantics: if for all functions in a string of functions applied to a value, $\eta = !$, R-AppS applies; if for all functions, $\eta = *$, R-AppP applies. For instance, reduction of $(\lambda x_1 {*}\langle\Box\rangle\,.\,\lambda x_2 !\langle\Box\rangle\,.\,x_1{\cdot}x_2)\; u_1{\cdot}u_2\; u_3{\cdot}u_4$ must go first through R-AppP, yielding $(\lambda x_2 !\langle\Box\rangle\,.\,u_1{\cdot}u_2{\cdot}x_2)\; u_3{\cdot}u_4$, and then through R-AppS, yielding $u_1{\cdot}u_2{\cdot}u_3{\cdot}u_1{\cdot}u_2{\cdot}u_4$.

3.3 The Numbering Relation

Analogous to the typing relation of the STLC [3], I introduce a *numbering relation* as a ternary relation on *number environments* Υ, terms t, and number specifiers π. I write $\Upsilon \vdash t \,\#\, \pi$ for an element of this relation and $\vdash t \,\#\, \pi$ if Υ is empty. Here (and analogously to type environments), a number environment $\varUpsilon$ is a mapping from variable names x to number specifiers π; I write $\varUpsilon, x \mapsto \pi$ for $\varUpsilon$ extended with the pair $x \mapsto \pi$.

Membership of $\varUpsilon \vdash t \,\#\, \pi$ in the numbering relation is derived by the following *numbering rules* (in which $\_ \to \_\,;\_$ is the conditional operator):

$$[\text{N-S}\textsc{tr}]\ \frac{n \neq 1 \qquad \left(\varUpsilon \vdash e_i \,\#\, \_\langle\mu\rangle\right)_{i=1}^{n}}{\varUpsilon \vdash e_1 \cdot \ldots \cdot e_n \,\#\, {*}\langle\mu\rangle} \qquad [\text{N-V}\textsc{ar}]\ \frac{\varUpsilon(x) = \eta\langle\mu\rangle}{\varUpsilon \vdash x \,\#\, \eta\langle\mu\rangle}$$

$$[\text{N-F}\textsc{un}]\ \frac{\varUpsilon, x \mapsto \eta\langle\mu\rangle \vdash t \,\#\, \eta'\langle\mu'\rangle}{\varUpsilon \vdash \lambda x \eta\langle\mu\rangle \,.\, t \,\#\, !\langle\mu \overset{\eta\,\eta'}{\rightharpoondown} \mu'\rangle} \qquad [\text{N-A}\textsc{pp}]\ \frac{\varUpsilon \vdash t_1 \,\#\, \eta_1\langle\mu_1\rangle \qquad \varUpsilon \vdash t_2 \,\#\, \eta_2\langle\mu_1 \overset{\eta_0\,\eta_3}{\rightharpoondown} \mu_2\rangle \quad \eta = \max((\eta_0 = * \to !\,; \eta_1), \eta_2, \eta_3)}{\varUpsilon \vdash t_2\ t_1 \,\#\, \eta\langle\mu_2\rangle}$$

$$[\text{N-C}\textsc{ns}]\ \varUpsilon \vdash c \,\#\, !\langle\square\rangle \qquad [\text{N-S}\textsc{ub}]\ \frac{\varUpsilon \vdash t \,\#\, !\langle\mu\rangle}{\varUpsilon \vdash t \,\#\, {*}\langle\mu\rangle}$$

For instance, for $\mathit{concat} = \lambda x_1{*}\langle\square\rangle \,.\, \lambda x_2{*}\langle\square\rangle \,.\, x_1{\cdot}x_2$, we have

$$\vdash \mathit{concat} \,\#\, !\langle\square \overset{*\,!}{\rightharpoondown} \square \overset{*\,*}{\rightharpoondown} \square\rangle \qquad \vdash \mathit{concat}\ c_1{\cdot}c_2 \,\#\, !\langle\square \overset{*\,*}{\rightharpoondown} \square\rangle \qquad \vdash \mathit{concat}\ c_1{\cdot}c_2\ c_3{\cdot}c_4 \,\#\, {*}\langle\square\rangle$$

and indeed, $\mathit{concat}\ c_1{\cdot}c_2\ c_3{\cdot}c_4 \longrightarrow (\lambda x_2{*}\langle\square\rangle \,.\, c_1{\cdot}c_2{\cdot}x_2)\ c_3{\cdot}c_4 \longrightarrow c_1{\cdot}c_2{\cdot}c_3{\cdot}c_4$.

If $\vdash t \,\#\, \pi$ for some π, I say that t is *well-numbered.* The following is of note:

- By N-Str, ϵ has number $*$; however, its mapping constraint μ remains unspecified (ϵ is polymorphic in a sense).
- In combination with N-Fun, N-Str makes sure that in a string of functions with length greater 1, all formal parameters x have the same number η (either ! or $*$).
- N-App makes the number of the formal parameter x of the applied function(s), η_0, decide (via the choice $\eta_0 = * \to !\,; \eta_1$) whether the number of the argument t_1, η_1, of an application $t_2\ t_1$ affects the number of the result of the application: if $\eta_0 = *$, the reduction must be through R-AppP, which means that the number of the argument is insignificant. For instance, $\vdash (\lambda x{*}\langle\_\rangle.c)\ v \,\#\, !\langle\square\rangle$, independently of the number of v.
- Nothing in N-App enforces that the number of the argument, η_1, and the number of the formal parameter, η_0, match. For $\eta_0 = !$, this is rendered unnecessary by R-AppS, which substitutes only elementary values for x (hence my choice of call-by-value); for $\eta_0 = *$, $\eta_1 = !$ is actually acceptable: an elementary value may always be substituted for a variable constrained to hold any number (but note how $\eta_1 = !$ does not propagate through a function: e.g., $\vdash (\lambda x{*}\langle\square\rangle \,.\, x)\ c \,\#\, {*}\langle\square\rangle$).
- N-Sub makes the number system polymorphic: all terms having number ! also have number $*$. For number judgements $\varUpsilon \vdash t : !\langle\mu \overset{\eta\,!}{\rightharpoondown} \mu'\rangle$ (derived through N-Fun, i.e., for functions), this means that we also have $\varUpsilon \vdash t : !\langle\mu \overset{\eta\,*}{\rightharpoondown} \mu'\rangle$ and $\varUpsilon \vdash t : {*}\langle\mu \overset{\eta\,*}{\rightharpoondown} \mu'\rangle$: functions "to one" are subsumed by functions "to any". This gives us well-numbered heterogeneous function strings, i.e., strings whose elementary functions' mapping constraints vary between $\mu \overset{\eta\,!}{\rightharpoondown} \mu'$ and $\mu \overset{\eta\,*}{\rightharpoondown} \mu'$.
- Last but not least, the number system supports the labelling of well-numbered terms $\lambda x!\langle\mu\rangle \,.\, t$ as *total functions* or *relations*: if $\vdash \lambda x!\langle\mu\rangle \,.\, t \,\#\, !\langle\mu \overset{!\,!}{\rightharpoondown} \_\rangle$, then we may call $\lambda x!\langle\mu\rangle \,.\, t$ a *total function* (because it maps an elementary value to an elementary value); if $\vdash \lambda x!\langle\mu\rangle \,.\, t \,\#\, !\langle\mu \overset{!\,*}{\rightharpoondown} \_\rangle$, then we may think of $\lambda x!\langle\mu\rangle \,.\, t$ as a *relation* (because it may map an elementary value to any number of elementary values[5]). Note that by this definition and by the polymorphism introduced through N-Sub, all total functions are also relations.

[5] including the same value more than once, giving us a multi-relation; also, unlike for set-theoretic relations, the values are ordered (but note that both can be had in some relational database systems, and may indeed be desirable in certain domains)

3.4 Number Safety

As for typing, we want to be sure not only that the well-numberedness of a term guarantees that it can be reduced to a value, but also that the term's derived number correctly abstracts from the length of that value. This is expressed by the following theorem:

▶ **Theorem 1** (Number Safety). *If for a term t and some π, $\vdash t \,\#\, \pi$ and $t \xrightarrow{*} t'$ and there is no t'' so that $t' \longrightarrow t''$, then t' is a value and $\vdash t' \,\#\, \pi$.*

Proof. The proof, which follows a standard layout (see, e.g., [3]), follows immediately from proofs of progress and preservation, which are found in a companion report [7]. ◀

4 Supporting the Relations Language

While the operational semantics of the relations language relies on (flat) bags for representing multiple values (see Section 1 and also [1, 2]), the SNLC builds on strings, which are inherently ordered. This difference, which is owing to the syntactic nature of the LC (and the fact that syntax is not commutative), affects the equality of multiples, which I did not cover[6]. Leaving this fundamental difference aside, the SNLC provides a broad basis for Harkes and Visser's relations language.

4.1 Computing with Multiples

The SNLC's choice of singular and plural semantics of function application supports both distributing operations over and aggregations of multiple operands, as required by addition and summing of the relations language. To see this, assume that elementary integer addition, $+$, is a primitive of the SNLC whose use is numbered by the rule

$$[\text{N-A}\textsc{dd}]\ \frac{\varUpsilon \vdash t_1 \,\#\, !\langle\square\rangle \qquad \varUpsilon \vdash t_2 \,\#\, !\langle\square\rangle}{\varUpsilon \vdash t_1 + t_2 \,\#\, !\langle\square\rangle}\ .$$

Distributive Application. We can then define

$$\oplus \;=\; \lambda x_1!\langle\square\rangle \,.\, \lambda x_2!\langle\square\rangle \,.\, x_1 + x_2 \qquad (\text{with } \vdash \oplus \,\#\, !\langle\square \xrightarrow{!\,!} \square \xrightarrow{!\,!} \square\rangle)$$

as the distribution of elementary addition over strings, giving us (in infix notation) $1{\cdot}2 \oplus 3{\cdot}4 \xrightarrow{*} 4{\cdot}5{\cdot}5{\cdot}6$, which corresponds to $\{\!|\, x_1 + x_2 \mid x_1 \in \{\!|\, 1, 2 \,|\!\}, x_2 \in \{\!|\, 3, 4 \,|\!\} \,|\!\}$, the result of the same addition in the the relations language (see Section 1 and [1, 2]). At the same time, above definition of $\oplus$ (in concert with R-AppS on which the reduction of its application relies) gracefully handles the absence of numbers in the style of an `Optional` type: e.g., $1 \oplus \epsilon \xrightarrow{*} \epsilon$ (the strictness of R-AppS on ϵ).

Aggregation. As noted in Section 1, aggregation cannot be defined distributively, but requires holistic treatment of multiples. Rather than offloading this treatment entirely to a semantic domain (as the definition of the relations language did for its aggregation functions), we can implement aggregation – with the help of (explicit) state – using a combination of

[6] But note how this problem parallels that of the equality of lambda terms, which is subject to α-equivalence.

functions with plural and singular semantics, where the singular semantics does the necessary looping. For instance, replacing variable substitution with a mutable variables store and assuming variable assignment, we can define

$$\begin{aligned}\mathrm{sum} \;&= (\lambda x_0!\langle\Box\rangle\,.\,\lambda x_1{*}\langle\Box\rangle\,.\,(\lambda x_2{*}\langle\Box\rangle\,.\,x_0)\ ((\lambda x_3!\langle\Box\rangle\,.\,x_0 := x_0 + x_3)\ x_1))\ 0\\ &\qquad (\text{with} \vdash \mathrm{sum}\,\#\,!\langle\Box\overset{*\,!}{\rightarrow}\Box\rangle)\end{aligned}$$

where applying $\lambda x_3!\langle\Box\rangle\,.\,x_0 := x_0{+}x_3$ (singular semantics) to x_1 lets x_3 loop over the elements of x_1 (whose number specifier is $*\langle\Box\rangle$) and which gives us, for instance, sum $1{\cdot}2 \xrightarrow{*} 3$. To implement aggregation without resorting to state, we would need to introduce recursion and string deconstruction to the SNLC, which would let strings appear as built-in lists (but see Section 5 for why they are not).

4.2 Relations and their Navigation

The relations language caters for the declaration and navigation of n-ary, non-updatable relations. In the SNLC, I model these relations as extensionally specified (tabular) functions, and their navigation as function application.

Binary Relations. To support the representation and navigation of relations as sets of pairs (rather than computable functions, or λ-abstractions), I extend the SNLC with a new form of terms,

$$\text{case } t \text{ of } \overline{u : u}\ .$$

Here, $\overline{u : u}$ is a string of pairs of elementary values that can be viewed as an extensionally defined binary relation (or a two-column table), and t is a term that selects from the relation a string of values, namely the string of right members of pairs whose left members are matched by t. This behaviour is accomplished by the rules

$$[\text{R-Case1}]\ \frac{t \longrightarrow t'}{\text{case } t \text{ of } \overline{u : u'} \longrightarrow \text{case } t' \text{ of } \overline{u : u'}}$$

$$[\text{R-Case2}]\ \frac{\big(v_i = (u = u_i \rightarrow u'_i\,;\epsilon)\big)_{i=1}^{n}}{\text{case } u \text{ of } \Pi_{i=1}^{n}(u_i : u'_i) \longrightarrow \Pi_{i=1}^{n} v_i}$$

reducing a case expression to a string $v_1{\cdot}\ldots{\cdot}v_n$, of which each v_i is either u'_i or ϵ, depending on whether the left member u_i of a pair $u_i : u'_i$ equals the selector value u. This reduction behaviour of case expressions is abstracted by the number rule

$$[\text{N-Case}]\ \frac{\Upsilon \vdash t\,\#\,!\langle\mu'\rangle \qquad \big(\Upsilon \vdash u_i\,\#\,!\langle\mu'\rangle\big)_{i=1}^{n} \qquad \big(\Upsilon \vdash u'_i\,\#\,!\langle\mu\rangle\big)_{i=1}^{n}}{\Upsilon \vdash \text{case } t \text{ of } \Pi_{i=1}^{n}(u_i : u'_i)\,\#\,{*}\langle\mu\rangle}\,,$$

expressing that

- the selector term t needs to reduce to precisely one value whose associated mapping constraint μ' must match the mapping constraints of all first places of the pairs, and that
- the case expression reduces to a string of arbitrary length, whose elements all share the same mapping constraint μ (enforced by the third condition of the rule).

Navigation. Given case expressions and their reduction rules, the relations that they capture can be navigated by abstracting from the selector term and applying the resulting abstraction (function) to the source of the navigation. For instance, for

$$\mathit{children} \;=\; \lambda x!\langle\Box\rangle\,.\,\mathrm{case}\ x\ \mathrm{of}\ (c_1\!:c_2)\cdot(c_3\!:c_4)\cdot(c_3\!:c_5) \qquad (\text{with } \vdash \mathit{children}\,\#\,!\langle\Box\overset{!\;*}{\rightharpoondown}\Box\rangle)$$

we get

$$\mathit{children}\ c_1 \xrightarrow{*} c_2 \qquad \mathit{children}\ c_2 \xrightarrow{*} \epsilon$$
$$\mathit{children}\ c_3 \xrightarrow{*} c_4\cdot c_5 \qquad \mathit{children}\ c_1\cdot c_2\cdot c_3 \xrightarrow{*} c_2\cdot c_4\cdot c_5\ .$$

Note how this corresponds to a navigation expression `x.children` with `x` substituted accordingly; in fact, if *children* is interpreted as the relation (or mapping) $\{(c_1,c_2),(c_3,c_4),(c_3,c_5)\}$, then its application to, say, $c_1\cdot c_2\cdot c_3$ can be interpreted as the direct image of the set $\{c_1,c_2,c_3\}$ under that relation. Also, relations can be navigated transitively: with

$$\mathit{children} \;=\; \lambda x!\langle\Box\rangle\,.\,\mathrm{case}\ x\ \mathrm{of}\ (c_1\!:c_2)\cdot(c_2\!:c_3)\ ,$$

we get

$$\mathit{children}\ (\mathit{children}\ c_1) \xrightarrow{*} c_3\ ,$$

corresponding to the navigation expression `c₁.children.children`.

Bidirectional Navigation. The encoding of relations using abstractions over case expressions is directed (the relations of the SNLC are mappings). In order to change the direction of navigation of a relation, or to make relations bidirectional as in the relations language, one needs to define relations in pairs, one per direction. Since these pairs are symmetric (one is the permutation of the other), it is straightforward to generate them using suitable preprocessing of SNLC programs.

Attributes. The relations language has not only relations relating objects, but also attributes describing them. In the SNLC, these attributes are modelled as special relations: if constants c are divided into objects, o, and attribute values, a, (integers, booleans, etc.), then abstractions *attr* of the form $\lambda x!\langle\Box\rangle\,.\,\mathrm{case}\ x\ \mathrm{of}\ \overline{o\!:a}$ associate objects with their attribute values (one abstraction *attr* per attribute). For instance, given a definition of the attribute *age*, we can write

$$\mathit{age}\ (\mathit{children}\ x)\ ,$$

corresponding to the expression `x.children.age` from Section 1.

Higher-Degree Relations. The relations language also caters for ternary and higher-degree relations. In the SNLC, one can model such relations by nesting (abstracted) case expressions: for instance,

$$R \;=\; \lambda x_1!\langle\Box\rangle\,.\,\mathrm{case}\ x_1\ \mathrm{of}\ c_1\!:\big(\lambda x_2!\langle\Box\rangle\,.\,\mathrm{case}\ x_2\ \mathrm{of}\ (c_2\!:c_3)\cdot(c_2\!:c_4)\big)$$

corresponds to a ternary relation $\{(c_1,c_2,c_3),(c_1,c_2,c_4)\}$, which is navigated by applying R to two values in a row: for instance, $R\ c_1\ c_2 \xrightarrow{*} c_3\cdot c_4$.

Higher-Order Relations. Given that case expressions correspond to relations, nested case expressions like those of R above correspond to nested relations, or relations of relations, which are by definition higher-order.

We can also define higher-order relations computationally. For instance, the definition

$$\mathit{hop}_2 \;=\; \lambda x_1!\langle\Box\xrightarrow{!\,*}\Box\rangle\,.\,\lambda x_2!\langle\Box\rangle\,.\,x_1\ (x_1\ x_2)$$

lets us rephrase the above term *children* (*children* c_1) as hop_2 *children* c_1. Assuming a second relation *parents*, we can derive both the grandchildren and the grandparents of c_1 using hop_2 *children·parents* c_1 (which reduces to (*children* (*children* c_1))·(*parents* (*parents* c_1))). If we wanted to navigate to siblings (children of parents) and spouses (parents of children) of c_1 as well, we would need to switch hop_2 to plural semantics (so that hop_2 *children·parents* c_1 reduced to *children·parents* (*children·parents* c_1)).

hop_2 composes a relation with itself. More generally, one might want to define relation composition (or relative multiplication) as a higher-order relation

$$\mathit{compose} \;=\; \lambda x_1!\langle\Box\xrightarrow{!\,*}\Box\rangle\,.\,\lambda x_2!\langle\Box\xrightarrow{!\,*}\Box\rangle\,.\,\lambda x_3!\langle\Box\rangle\,.\,x_1\ (x_2\ x_3)\ ,$$

but for *compose* to be fully general (i.e., applicable to arbitrarily numbered relations), one would need to adopt parametric number specifiers (in analogy to parametric types).

4.3 More Multiplicities

The relations language features not two, but four different numbers (there called multiplicities): besides ! and $*$, which are also covered by the SNLC as presented here, it offers ? (for “none or one”) and + (for “one or more”). While integrating these in the SNLC will require some extra effort (the proofs will require significantly more case analyses), this effort may be well-spent: rather than the number η in the derived number specifier of $\lambda x!\langle\mu\rangle\,.\,t$, $!\langle\mu\xrightarrow{!\,\eta}\_\rangle$, distinguishing between total functions ($\eta = !$) and relations ($\eta = *$), it qualifies $\lambda x!\langle\mu\rangle\,.\,t$ as a relation with properties given by the table

η	left-total	right-unique	classification
?	no	yes	partial function
!	yes	yes	total function
$+$	yes	no	relation
$*$	no	no	relation

Furthermore, restricting numbers η to ! and ?, we arrive at a calculus that can distinguish and handle total and partial functions without needing an `Optional` type (and a type system supporting it): for instance, an implementation of subtraction ($\ominus$) may have number specifier $!\langle\Box\xrightarrow{!\,!}\Box\xrightarrow{!\,?}\Box\rangle$, indicating that applying it to two arguments evaluates to nothing (ϵ) if the subtrahend is greater than the minuend. The expression $1 \ominus 2 \oplus 3$ would then evaluate to ϵ, without the definition of $\oplus$ needing to take extra measures for this (it is automatically strict on ϵ; see Section 4.1).

5 Discussion

One might hold against the SNLC that it makes strings a primitive language construct, where the plain LC is already expressive enough to cover strings, by giving them the form of lists. However, like sets and unlike strings, lists have a deep structure (one can have lists of lists) and indeed, applying a function to a string is not the same as mapping the function over the corresponding list: for instance $(\lambda x!\langle\Box\rangle\,.\,\epsilon)\ c_1{\cdot}c_2$ reduces to ϵ, a string of length 0, and not to a string of length 2, as mapping would do. It appears that at the very least, the flattening that is implicit in the concatenation of strings would need to be added to function application somehow. With this in place, however, and assuming a typing discipline without coercions, one would need to encode everything as a list, if only because the type of a list differs from that of its elements so that one element cannot stand in for any number, as suggested by N-Sub and realized by substitution in the SNLC. On the other hand, the SNLC's lack of both recursion and deconstruction of strings (analogous to the deconstruction of lists into their heads and tails) lets aggregation require explicit state, unlike the fold on lists that has granted functional programming much of its popularity. A more practical language based on the SNLC will therefore likely feature both, strings and lists (but mind that, being a heir to the LC, all that the SNLC is lacking to accommodate lists is recursion).

In several extensions of first-order languages with numbers (or multiplicities), type and number have been observed to be orthogonal (see, e.g., [1, 2, 5, 6]). By adding higher-order functions, however, it becomes apparent that the number annotations must have a form that is parallel to that of types: they must introduce "function numbers" $\xrightarrow{\eta\,\eta'}$ (here called mapping constraints) as analogues of function types. To terminate the recursion of mapping constraints, a single "unary mapping constraint", or "mapping constraint of a constant" must be introduced (here $\square$). While this is much like the one base type that is minimally required by any STLC to be usable [3], in the SNLC, there is actually no use in having more than one such base. Therefore, the terminal $\square$ can be replaced by syntax for (arbitrarily many) base types, which equips us for defining a "simply *typed and numbered* lambda calculus" as a straightforward merger of the STLC and the SNLC as here presented.

6 Conclusion

By adding strings of terms, I have extended the lambda calculus to a higher-order relational language in which an elementary function may map an elementary value to zero, one, or more elementary values. Rather than typing this language, I have numbered it, and have shown that this gives us guarantees analogous to those of typing, with the edge that the number system can distinguish between total functions and relations (ordered multi-relations, to be precise). It turns out that the SNLC serves existing, more complex query languages; specifically, it serves as a foundation of the relations language of Daco Harkes and the late Eelco Visser.

References

1 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2014. doi:10.1007/978-3-319-11245-9_14.

2 Daniël Corstiaan Harkes. *Declarative Specification of Information System Data Models and Business Logic*. PhD thesis, Delft University of Technology, Netherlands, 2019. doi:10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a.

3 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

4 Harald Søndergaard and Peter Sestoft. Non-determinism in functional languages. *Comput. J.*, 35(5):514–523, 1992. doi:10.1093/comjnl/35.5.514.

5 Friedrich Steimann. None, one, many – What's the difference, anyhow? In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 294–308. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.SNAPL.2015.294.

6 Friedrich Steimann. Containerless plurals: Separating number from type in object-oriented programming. *ACM Trans. Program. Lang. Syst.*, 44(4), September 2022. doi:10.1145/3527635.

7 Friedrich Steimann. A simply numbered lambda calculus with number safety proof. Report, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, Sept 2022. URL: https://feu.de/ps/pubs/SNLC-TR.pdf.

8 Friedrich Steimann and Marius Freitag. The semantics of plurals. In Bernd Fischer, Lola Burgueño, and Walter Cazzola, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*, pages 36–54. ACM, 2022. doi:10.1145/3567512.3567516.

The Ultimate GUI Framework: Are We There Yet?

Knut Anders Stokke[1]
University of Bergen, Norway

Mikhail Barash
University of Bergen, Norway

Jaakko Järvi
University of Turku, Finland

Abstract

The programming community seems to be forever searching for the ultimate user interface programming approach and the accompanying framework. We describe the landscape of recent efforts in this quest through describing commonalities and differences of modern JavaScript frameworks with respect to their approaches to GUI specifications. We situate both Eelco Visser's work on WebDSL and our own work on GUI programming in this landscape, and point out areas where more research is needed, including modeling multi-way dataflows and dynamic structures in GUIs.

2012 ACM Subject Classification Software and its engineering → Graphical user interface languages; Software and its engineering → Software libraries and repositories

Keywords and phrases Web frameworks, GUI specification, WebDSL, HotDrink, WarmDrink

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.25

Acknowledgements The authors are thankful to the anonymous reviewers for their valuable comments on the paper. We also wish to express our gratitude for and admiration of Eelco Visser's life's work on programming research, and of him as a person.

1 Introduction

Since the introduction of the Model-View-Controller pattern [32], the programming community has been chasing after better ways to implement graphical user interfaces (GUIs) – to arrange how events are delivered and handled and how source code related to GUIs is organized. We have seen many new patterns [29, 18, 14] and a large number of *frameworks* (we will list many in this paper) over the years. New GUI patterns and frameworks have at times come with some hype, and have been met with excitement that the new way to program GUIs is not merely better, but rather the best, or even the ultimate way of programming GUIs. Such excitement has gradually faded with the introduction of new ultimate GUI frameworks.

How far have we come since 1979, when MVC was proposed? Are we close to the ultimate framework, the final truth in GUI programming? Or are there still features and aspects of GUI programming that can be improved, even significantly? We claim the answer to the last question is positive, and point to at least two such areas.

First, none of the widely used modern GUI frameworks support complex *data dependencies* between variables, which occur in *data-rich* user interfaces. An example of this is a dialog for image resizing: editable variables include the absolute width and height in pixels, the relative width and height in percentages, and the ratio. The program must maintain several relations between all these variables whenever a user updates one of them. Many modern GUI frameworks support functional, but not relational (*multi-way*), dependencies between

[1] Corresponding author.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 25; pp. 25:1–25:9
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

variables. When application programmers decompose relational dependencies into functional and manage themselves which dependencies should be in effect, GUI logic tends to become scattered throughout event handling functions [22].

Second, GUIs present all kinds of *structures* (such as lists, grids, and trees) to users. There are standard operations for manipulating such structures that users would like to take advantage of, but often GUIs do not provide such tools to the user. Our oft-used example is *ApplyTexas*[2], a website handling admission applications to higher education institutions in the State of Texas. This GUI asks users to provide a list of extracurricular activities, each consisting of more than 20 input fields, in the order of importance, but offers no operations to reorder activities [16]. Modern GUI frameworks lack direct support for generic reusable structure manipulation operations; providing such operations is the responsibility of the application programmer.

This paper surveys a large number of recent popular GUI frameworks and forms a landscape of the different approaches and essential features that can be identified in these frameworks. Further, it argues that the (lack of) quality in today's GUIs is an indication that the ultimate GUI framework or GUI programming paradigm has not yet been discovered, and then discusses challenges, focusing on the two we mentioned above, of GUI programming that today's popular frameworks do not provide answers for. We also briefly describe our own efforts in addressing those challenges and relate our work to Eelco Visser's WebDSL [20], which is a collection of domain-specific languages and tools for developing web applications.

2 The Landscape of JavaScript GUI Frameworks

Frameworks provide a standardized way of developing software through *inversion of control* [8]: the program's control flow is dictated by the framework and not the programmer, who only provides relevant code to get behaviour specific to the application at hand. Frameworks separate the data layer (the "*model*") from the presentation layer (the "*view*") and define how the two interact with each other; this is known as *separation of concerns* [19]. Client-side web frameworks exhibit different approaches to this, and these approaches dictate how user interfaces are *specified*. Usually the programmer specifies the view declaratively, with references to the model's variables. When the variables change, these views are dynamically updated. However, the approaches to update the views and track variable changes differ between frameworks.

To capture a *landscape* of modern GUI frameworks, we gather properties and features where these frameworks manifest different design choices, and characterize each framework through these properties. The cross-tabulation of the properties and frameworks is given in Table 1. The set of frameworks we included is not an exhaustive collection of all JavaScript frameworks; our goal was to include the influential and popular frameworks that have appeared since the early 2010's, but of course the selection is subjective[3].

The common task of all these frameworks is to keep an application's view in sync with its model. The *model* is a collection of data that users manipulate through the user interface, and, for web applications, the *view* is the website's Document Object Model (DOM). Frameworks that update the DOM automatically on model changes are called *reactive*.

[2] `https://www.applytexas.org/`

[3] Certainly, there are other relevant GUI frameworks worth analysing in our setting that are based on different languages than JavaScript, most prominently perhaps SwiftUI [39] and Flutter [10].

Table 1 A comparison of widely used JavaScript frameworks, the constraint system-powered library HotDrink, WebDSL, and the properties that these support. A fully supported property is denoted by ●, a partially supported property is denoted by ◒, and non-supported properties are denoted by ○. The property columns have the following meaning: *declarative view specification* means that views are specified declaratively; *re-rendering mechanism* specifies the approach used to re-render the view; *imperative rendering* means that DOM elements are updated imperatively; *virtual DOM* means that a virtual DOM is used to identify which elements need to change; *compiled rendering* means that a view specification is used to generate code for updating DOM elements; *view replacement* means that on re-render, the entire component view is replaced by a new one, instead of being updated; *two-way binding* means that such bindings are supported; *stateful components* means that components have self-contained state; *component hierarchy* means that components can have sub-components and pass data to them; *multi-way dataflow* means that the reactive system supports multi-way dataflow; *consistent concurrency* means that the reactive system handles asynchronous computations consistently [11]; *semantic model* means that high-level abstract GUI structures are defined; *tracking mechanism* specifies the approach used to track model updates; *setter function* means that the model is updated through a setter function; *messages (MVU)* means that GUI events produce messages to a global update function; *observables* means that model attributes are wrapped in observables (tracked objects) that notify subscribers on value change; *checked after events* means that on every event handling, the model is checked for changes; *compiled tracking* means that model updates are compiled to instructions that notify the system after the update.

Property / Framework	Declarative view spec. [23]	Re-rendering mechanism				Two-way bindings [43]	Stateful components [35]	Component hierarchy [42]	Multi-way dataflow [21]	Consistent concurrency [11]	Semantic model [13]	Tracking mechanism				
		Imperative rendering	Virtual DOM [44]	Compiled rendering [28]	View replacement							Setter-function [35]	Messages (MVU) [6]	Observables [17, 26]	Checked after events	Compiled tracking [1]
Angular [2]	●	○	○	●	○	●	●	●	○	○	○	○	○	○	●	○
Backbone.js [3]	◒	●	○	○	●	○	●	○	○	○	○	●	○	○	○	○
Elm [9]	●	○	●	○	○	○	○	●	○	○	○	○	●	○	○	○
Ember.js [7]	●	○	○	○	●	●	●	●	○	○	○	○	○	●	○	○
Knockout.js [25]	●	●	○	○	○	●	●	●	○	○	○	○	○	●	○	○
React [30]	●	○	●	○	○	○	●	●	○	○	○	●	○	○	○	○
SolidJS [34]	●	○	○	●	○	○	●	●	○	○	○	●	○	●	○	○
Svelte [38]	●	○	○	●	○	●	●	●	○	○	○	○	○	○	○	●
Vue.js [45]	●	○	●	○	○	●	●	●	○	○	○	○	○	●	○	○
HotDrink [15, 36, 37]	◒	●	○	○	○	●	●	●	●	●	●	○	○	●	○	○
WebDSL [20]	●	n/a	n/a	n/a	n/a	●	○	●	○	○	●	n/a	n/a	n/a	n/a	n/a

Backbone.js [3] is one of the oldest among the compared frameworks. As most of the frameworks discussed in this section, it enables programmers to build user interfaces from *components* that are rendered individually. For each component, one needs to implement a rendering function that renders the component using attributes from the model; on model changes, the component is re-rendered and the old DOM is replaced with the new one. Such a rendering mechanism is different from the approach of Angular [2], where component specifications are *compiled*. During the compilation process, the framework derives how each

DOM element depends on the model attributes, and generates functions to update only those elements that need to change. After an event (e.g., a mouse click, a keypress, etc.) has been handled – possibly resulting in updates to the model – Angular compares the current model with the old one and updates the corresponding DOM elements.

Most JavaScript frameworks support one-way bindings in components' view specifications: these bindings update DOM-elements on model changes. Angular, however, enables *two-way bindings* [43]. Such a binding between a model attribute and a user-editable value (e.g., the text in an input field, or a slider's position) means that the framework keeps the two in sync. In frameworks without support for two-way bindings, event handlers are needed to update the model when a user modifies values in the view. In addition to Angular, two-way bindings are supported in the Ember.js [7] and Knockout [25] frameworks. The former uses the design philosophy of *convention over configuration* [46] by providing built-in code for common web application functionality.[4] The latter uses the model-view-viewmodel architecture [9] in order to separate domain data (the *model*), the *view*, and data presented and edited in the view (the *viewmodel*). Knockout treats the (view)model differently than Angular and Backbone.js: instead of using plain JavaScript objects, it wraps data in *observable* objects, which notify their *observers*[5] whenever their value gets updated. That is, when a DOM-element is bound to an observable (view)model attribute, an observer updates the DOM-element on change notification. This distinguishes Knockout from other frameworks that have to compute the changes to the model – or, for that matter, to the view – after every (user) event has been handled.

An entirely different approach to GUI specifications is used in Elm [9], which is a *purely functional* Haskell-like domain-specific language (rather than a framework). Elm introduces the model-view-update (MVU) architecture [14], where the model is an immutable data structure and the view is a function that maps this structure to a DOM. An Elm application has an update function which is called every time an event is fired. This function takes a model and an event as arguments, and it produces a new updated model, which is further used to re-render the view. This architecture can be utilized by other frameworks, such as React [30]. Elm and React, together with another JavaScript framework Vue.js [45], use a rendering method that constructs a new *virtual* DOM each time an update to the model has been made. This new DOM is then *reconciled* with the current DOM [27], that is, the new and the old views are compared in order to find the least number of edits to the view needed to reflect the updated model. After the comparison is made, the necessary modifications are introduced into the actual DOM. This is different from other frameworks (such as Backbone.js), which replace the *entire* current DOM with an updated one. Another distinct feature of Elm is the way it handles *modularity*: rather than being composed from components, GUIs are only specified with pure functions. The entire application model is persisted in one immutable data structure, and all events are handled in a single update function. This is in contrast to React and Vue.js, where state is maintained at the component level and not at the application level[6]. Frameworks Angular, Svelte [38], and SolidJS [34] similarly handle state changes at the component level, but compile component specifications to lower-level JavaScript, and perform static analysis of GUI specifications in order to find dependencies between DOM elements and model attributes. With Svelte, for instance, when a component state attribute is referenced in a component's view specification, the

[4] This is similar to the approach of the back-end framework Ruby on Rails [40].

[5] An observer is a function that takes one argument, the updated value, and performs a side-effect with it.

[6] With the *Redux* [31] extension, state can be handled at the application level also in React.

compiled component code has references to the component's DOM elements and has an update-function; the function takes (updated) component states as argument and updates the state attribute into the corresponding DOM element [28].

3 Multi-Way Dataflow and Structural Modifications

One of the early use cases for JavaScript was to enable *dynamic* client-side changes on websites [48]. While web-servers could construct HTML documents dynamically on request, JavaScript could run on the client-side, react to user interaction, and modify content without refreshing the page, thus enabling a more interactive user experience on the web. JavaScript scripts can show validation messages to users while they type into a text field, update widgets as a user moves a slider, give up-to-date information about the weather forecast as the user types in a city name, and so on. All changeable GUI values can be considered as *variables*, with dependencies between them that should be maintained whenever there is a change to any of the variables. Reacting to such changes is traditionally done in *event handlers* that are triggered by user events; maintaining dependencies directly in event handlers leaves a large coordination effort to the application programmer: *all* event handlers become responsible for *all* GUI variables, and introducing new GUI variables or relations may necessitate modifications to several event handlers. Event handlers with asynchronous tasks require even more involved modifications: before updating a variable, such a handler must ensure that the variable is not currently used as input to an asynchronous computation, in order to keep the GUI in a consistent state [12].

Reactive programming [5, 47] simplifies maintaining dependencies between GUI variables. In this programming model, an underlying system is made aware of all variable dependencies in the GUI, so that every variable change can be propagated to all variables that depend on the changed variable. Most frameworks discussed in Section 2 have some support for reactive programming [3, 25, 2, 7, 9, 30, 45, 38, 34], but they only support one fixed dataflow. While this does not constitute a problem in GUIs where it is known in advance which variables are fixed and which need to be computed, data-rich user interfaces often require support for multiple ways to update variables.

As a familiar example, consider a form for booking a hotel room. The form has fields for the arrival and departure dates, and the number of nights. Some users may want to fill in the first two fields and have the last one computed, while others may prefer to provide the arrival date and number of nights, and have the departure date computed for them. Many hotel-booking applications support only one fixed dataflow, and show some of the variables as non-editable GUI elements (if showing them at all). A possible reason for the prevalence of such feature-limited user interfaces is the accidental complexity [4] of implementing and orchestrating multiple dataflows by hand.

An alternative to maintaining dependencies manually is to define them as constraints in a *constraint system*. HotDrink [15] is an example of a *multi-way dataflow constraint system*-powered library that enables specifying relations, multi-way dependencies, between variables. Each constraint in such a specification has satisfaction methods that enforce the constraint when executed. Whenever a variable is updated, the library solves the constraint system [33] by choosing and executing one method from each constraint, in order to satisfy all the relations between the variables. In addition, the history of the last edited values is maintained, and is used to find dataflows that are least "surprising" to the user [12].

Listing 1 specifies a constraint component for the hotel booking example using the HotDrink DSL. The first constraint represents the relation that `nights` is the number of nights between `arrival` and `departure`, and has three satisfaction methods to enforce

Listing 1 A HotDrink specification of multi-way dependencies in a GUI.

```
component HotelBooking {
  var arrival, departure, nights, showErrorMessage;
  constraint {
    (arrival, departure -> nights) =>
      Math.ceil((date2.getTime()-date1.getTime())/(1000*3600*24));
    (arrival, nights -> departure) => ...;
    (departure, nights -> arrival) => ...;
  }
  constraint {
    (nights -> showErrorMessage) => nights <= 0;
  } }
```

this constraint, thus enabling all three variables to be set by a user. The fourth variable, `showErrorMessage`, is a boolean variable that is true if a stay would have zero or a negative number of nights, as specified in the last constraint (this variable can not be set by the user). A user interface with input fields for specifying hotel bookings can be connected with this component (with two-way bindings between input fields and variables) to ensure that the relation between the fields is maintained. Additionally, the view can subscribe to the observable variable `showErrorMessage` and use it to toggle the display of an error message on invalid booking dates.

The core Hotdrink library allows the programmer to specify constraint system *components*, collections of variables and constraints. Further, it has a (low-level) API for linking components via sharing of variables. Implementing structural operations using this API is, however, error-prone, as operations must handle many different scenarios. For instance, in a list where each component is connected to the succeeding component, removing the first component involves different constraint system updates than removing the last component, or removing a component in between two others. And of course, if the component specification itself changes, structural operations might have to be modified.

An extended version of the library, called WarmDrink [36, 37], builds on these low-level operations and enables composing components into *structures*, such as lists and trees, with data flowing between the components. With WarmDrink the programmer specifies concisely how constraint components are related within a structure. Based on this specification WarmDrink generates a high-level API for modifying the GUI structure, e.g., an API for adding, removing and swapping connected components in lists. Equipped with such an API, the application programmer can easily provide a rich set of structure manipulation tools to the user.

4 Relating to WebDSL

The idea of generating full GUI implementations from concise high-level specifications of structure – similarly to what WarmDrink does – is notably implemented in *WebDSL* [20], which is a set of domain-specific languages for defining web applications, together with a static analysis tool that performs cross-language validation. These languages focus on defining entities (i.e., the *model*), pages (i.e., the *view*), access control, and various actions (e.g., persisting data when users submit a form). The WebDSL's static analyser checks, for example, the existence of properties displayed in a GUI, or the existence of pages defined in access control specifications.

Though WarmDrink and WebDSL solve different problems, their goal is to reduce the accidental complexity of web development, especially of form-based web applications. Both frameworks maintain relations between model attributes: while HotDrink enables specifying constraints on the variables and WarmDrink enables specifying relations between GUI components, WebDSL enables specification of aggregation relationships between entities. Both HotDrink and WebDSL support two-way bindings (as do Knockout.js, Angular and Svelte). WebDSL takes such bindings one step further than other frameworks: pages can call method `persist` on an entity that a user has edited, and this updated entity will be saved in the application's database. This conceptually binds the values of DOM elements to values persisted on a server.

As with most other frameworks discussed in this paper, when using WebDSL the view is specified with declarative HTML-templates, rather than by constructing DOM-elements with JavaScript code. A major departure from other frameworks is, however, that WebDSL renders web pages on the server, while the frameworks perform rendering in the browser. The latter approach enables web pages to update dynamically when the user navigates between them, but it also involves downloading the framework source code and starting the framework runtime in the browser, which has significant overhead compared to viewing the content of a plain HTML file.[7] Because of the overhead of client-side rendering, a recent trend is to run JavaScript frameworks on the server. This approach, known as *server-side rendering* [41], combines the expressiveness and modularity of JavaScript frameworks with the performance of server-side rendered websites, and also shifts JavaScript frameworks closer to WebDSL.

While data-rich web applications can be rendered on a server, users still do interact with input forms and widgets that affect each other's behaviour, and this behaviour should be updated *dynamically*. For instance, when filling out a form, users benefit from getting validation messages as they type, not after they have submitted the form. If the form additionally involves a *structure* of repeated fields, the GUI should provide operations for modyifying that structure. Observations today [37, 36] highlight that such crucial operations are often missing even in widely used web applications.

Many approaches and frameworks, WebDSL and our HotDrink and WarmDrink amongst them, have over the years made significant contributions to increase our understanding about GUI programming. We hope, however, that we have made it clear that there is (still) room and a need for more research on GUI programming. We are certainly continuing our investigations on multi-way dataflow constraint systems for GUIs, to move us closer to the, perhaps elusive, goal of "the ultimate GUI framework". Finally, while the WebDSL and JavaScript frameworks with server-side rendering enable embedding of JavaScript code into client-side HTML, making dynamic changes on a page, such code is typically self-contained and not statically checked against the rest of the source code. Perhaps addressing these issues could be a WebDSL-inspired avenue of inquiry towards that same goal.

References

1 Advanced Svelte: Reactivity, lifecycle, accessibility. Accessed 26.10.2022. URL: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Svelte_reactivity_lifecycle_accessibility.

[7] A blog post [24], surveying thousands of websites that use different JavaScript frameworks, reports that a median Angular website for mobile devices involves downloading 1.1 MB of JavaScript code and takes 4.1 seconds of scripting-related CPU time. For the 90th percentile, the numbers are 2.9 MB and 13.3 seconds, respectively.

2 Angular. Accessed 26.10.2022. URL: `https://angular.io/`.

3 Backbone.js. Accessed 26.10.2022. URL: `https://backbonejs.org/`.

4 Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. `doi:10.1109/MC.1987.1663532`.

5 Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997. `doi:10.1145/258949.258973`.

6 The Elm architecture. Accessed 26.10.2022. URL: `https://guide.elm-lang.org/architecture/`.

7 Ember.js – A framework for ambitious web developers. Accessed 26.10.2022. URL: `https://emberjs.com/`.

8 Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

9 Richard Feldman. *Elm in Action.* Manning, 2020.

10 Flutter – Build apps for any screen. Accessed 27.01.2023. URL: `https://flutter.dev/`.

11 Charles Gabriel Foust. *Guaranteeing Responsiveness and Consistency in Dynamic, Asynchronous Graphical User Interfaces.* PhD thesis, Texas A&M University, 2016.

12 Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. *SIGPLAN Not.*, 51(3):121–130, October 2015. `doi:10.1145/2936314.2814207`.

13 Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

14 Simon Fowler. Model-view-update-communicate: Session types meet the Elm architecture. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ECOOP.2020.14`.

15 John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: A library for web user interfaces. *SIGPLAN Not.*, 48(3):80–83, 2012. `doi:10.1145/2480361.2371413`.

16 John Alexander Freeman. *Reusable User Interface Behaviors.* PhD thesis, Texas A&M University, 2016.

17 Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.

18 Victor Gaudioso. MVVM: Model-view-viewmodel. In *Foundation Expression Blend 4 with Silverlight*, pages 341–367. Apress Berkeley, CA, 2010. `doi:10.1007/978-1-4302-2974-2_10`.

19 Danny Groenewegen, Zef Hemel, and Eelco Visser. Separation of concerns and linguistic integration in WebDSL. *IEEE Software*, 27(5):31–37, 2010.

20 Danny M. Groenewegen, Zef Hemel, Lennart C.L. Kats, and Eelco Visser. WebDSL: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, pages 779–780, New York, NY, USA, 2008. Association for Computing Machinery. `doi:10.1145/1449814.1449858`.

21 Magne Haveraaen and Jaakko Järvi. Semantics of multiway dataflow constraint systems. *Journal of Logical and Algebraic Methods in Programming*, 121:100634, December 2020. `doi:10.1016/j.jlamp.2020.100634`.

22 Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: From incidental algorithms to reusable components. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 89–98, New York, NY, USA, 2008. Association for Computing Machinery. `doi:10.1145/1449913.1449927`.

23 JSX. Accessed 26.10.2022. URL: `https://facebook.github.io/jsx/`.

24 Tim Kadlec. The cost of JavaScript frameworks, April 2020. Accessed 26.10.2022. URL: `https://timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/`.

25 Knockout. Accessed 26.10.2022. URL: `https://knockoutjs.com/`.

26 Knockout: Observables. Accessed 26.10.2022. URL: `https://knockoutjs.com/documentation/observables.html`.

27 Magnus Madsen, Ondrej Lhotak, and Frank Tip. A semantics for the essence of React. In *European Conference on Object-Oriented Programming*, 2020.

28 Joshua Nussbaum. The Svelte compiler: How it works, February 2020. Accessed 26.10.2022. URL: `https://dev.to/joshnuss/svelte-compiler-under-the-hood-4j20`.

29 Mike Potel. MVP: Model-view-presenter: The Taligent programming model for C++ and Java, 1996. URL: `www.wildcrest.com/Potel/Portfolio/mvp.pdf`.

30 React – A JavaScript library for building user interfaces. Accessed 26.10.2022. URL: `https://reactjs.org/`.

31 Redux – A predictable state container for JavaScript apps. Accessed 27.01.2023. URL: `https://redux.js.org/`.

32 Trygve Reenskaug. Models-views-controllers. Reproduced in The original MVC reports by University of Oslo, December 1979. URL: `https://www.duo.uio.no/bitstream/handle/10852/9621/Reenskaug-MVC.pdf`.

33 Michael Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, pages 137–146, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/192426.192485`.

34 SolidJS – Reactive JavaScript Library. Accessed 26.10.2022. URL: `https://www.solidjs.com/`.

35 State and lifecycle. Accessed 26.10.2022. URL: `https://reactjs.org/docs/state-and-lifecycle.html`.

36 Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. Manipulating GUI structures declaratively. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, pages 63–69, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3425898.3426956`.

37 Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. A domain-specific language for structure manipulation in constraint system-based GUIs. *Journal of Computer Languages*, 74:101175, 2023.

38 Svelte – Cybernetically enhanced web apps. Accessed 26.10.2022. URL: `https://svelte.dev/`.

39 SwiftUI – Apple Developer Documentation. Accessed 27.01.2023. URL: `https://developer.apple.com/documentation/swiftui/`.

40 Bruce Tate and Curt Hibbs. *Ruby on Rails: Up and Running.* O'Reilly Media, Inc., 2006.

41 Mohit Thakkar. *Building React Apps with Server-Side Rendering.* APress, Berlin, Germany, 1 edition, April 2020.

42 Thinking in React. Accessed 26.10.2022. URL: `https://reactjs.org/docs/thinking-in-react.html`.

43 Two-way binding. Accessed 26.10.2022. URL: `https://angular.io/guide/two-way-binding`.

44 Virtual DOM and internals. Accessed 26.10.2022. URL: `https://reactjs.org/docs/faq-internals.html`.

45 Vue.js – The progressive JavaScript framework. Accessed 26.10.2022. URL: `https://vuejs.org/`.

46 Irena Petrijevcanin Vuksanovic and Bojan Sudarevic. Use of web application frameworks in the development of small applications. In *2011 Proceedings of the 34th International Convention MIPRO*, pages 458–462. IEEE, 2011.

47 Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, May 2000. `doi:10.1145/358438.349331`.

48 Allen Wirfs-Brock and Brendan Eich. JavaScript: The first 20 years. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. `doi:10.1145/3386327`.

Refactoring = Substitution + Rewriting

Towards Generic, Language-Independent Refactorings

Simon Thompson[1]
School of Computing, University of Kent, Canterbury, UK
Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

Dániel Horpácsi
Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

Abstract

Eelco Visser's work has always encouraged stepping back from the particular to look at the underlying, conceptual problems.

In that spirit we present an approach to describing refactorings that abstracts away from particular refactorings to classes of similar transformations, and presents an implementation of these that works by substitution and subsequent rewriting.

Substitution is language-independent under this approach, while the rewrites embody language-specific aspects. Intriguingly, it also goes back to work on API migration by Huiqing Li and the first author, and sets refactoring in that general context.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases refactoring, generic, language independent, rewriting, substitution, API upgrade

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.26

Funding *Simon Thompson*: With support from the UK Engineering and Physical Sciences Research Council awards EP/N028759/1, EP/C524969/1, GR/R75052/01 and EP/T014512/1.
Dániel Horpácsi: With support from the NRDI Fund of Hungary and financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

1 Introduction

Our subject here is not new: Eelco Visser initiated a discussion of language-independence transformations [25] in the millennium year; it is a pleasure and an honour to join the conversation that he began.

Refactoring tools are a particularly sensitive part of a programmer's toolkit, since they can make large-scale modifications to code, and yet are expected not to change the observable behaviour of the system. Users therefore need to be given assurance about the safety of using a tool. Assurance generally comes in two complementary forms, through verification and through architecture.

Arguments can be made about the *correctness* of the transformations made. These can be black box, checking the original against the refactored code without examining how the transformation is performed. At minimum this is achieved by regression testing, but can be augmented by checking equivalence more generally [8, 9]. Looking inside the implementation it is also possible – at least in principle – to prove the correctness of that transformation [23].

[1] Corresponding author

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 26; pp. 26:1–26:9
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These checks will apply to particular *instances* of a refactoring in the case of regression testing, while verification should establish the correctness of the transformation in itself, that is for *all* possible instances. Testing can also be used in the latter case, e.g. by generating random transformations of an arbitrary system, and testing the "old" code against the "new" with random inputs [5].

An alternative source of assurance is in the *architecture* of the refactoring tool itself. At its heart, any refactoring tool works by transforming a complex data structure (AST or database) representing a program into a related structure. With no further thought, each refactoring can be constructed anew, by means of an *ad hoc* recursive function. But this can be improved. Firstly, following Eelco Visser's work [25], it is possible to take a higher-level view of the way the data structure is traversed, using a *strategic* approach in something like Stratego [4], Strafunski [15] and related systems.

Secondly, we can see a commonality between the refactorings themselves, and this is the approach we outline here. With this perspective, a class of refactorings can all be performed with a single implementation. This, in turn, reduces the burden on users wishing to assure themselves of the soundness of the implementation, and indeed has positive consequences for formal verification of the refactoring transformations too.

In the remainder, we first introduce the separation of generic and specific elements of refactoring definitions in §2. Then in §3 we argue that the generic parts in many cases can be made language-independent. Finally, §5 discusses some related work and §6 concludes.

2 Refactoring = Substitution + Rewriting

A class of refactorings in Erlang are concerned with the definition and subsequent use of functions, including, among others, renaming and generalisation. In this section we show how these can be described in a common way, and the implications of this for verification.

2.1 Function renaming

Consider the example of function renaming.

Before renaming

```
f(X) -> X+1.
g(Y) -> f(Y+2) - f(Y-2).
```

After renaming

```
h(X) -> X+1.
g(Y) -> h(Y+2) - h(Y-2).
```

The transformation is described by showing how the function *definition* is changed, and also explaining how to change each *use* of the function. We do that thus:

Define the modified function:

```
h(X) -> X+1.
```

Implement the old function using the new:

```
f = fun(X) -> h(X) end
```

How does this describe the refactoring? We can use the implementation of the old function in terms of the new to give us the new code, in a series of steps, thus:

```
g(Y) -> f(Y+2) - f(Y-2).
-- by substitution giving
g(Y) -> (fun(X) -> h(X) end)(Y+2) - (fun(X) -> h(X) end)(Y-2).
-- and by rewriting (beta-reduction, here)
g(Y) -> h(Y+2) - h(Y-2).
```

Rewriting stops at this point: we don't want, or need, to inline `h`, since we want to be faithful to the original program that contains a function call to `f` here.

It is important to note that this approach works for other uses of the function `f`, including as `fun` arguments to higher-order functions, and, with some preliminary eta-expansion [2], in calls to `spawn` the function.

2.2 Function generalisation

Now we look at a second example, function generalisation, and see that it fits the same pattern.

Before generalisation

```
f(X) -> X+3.
g(Xs) -> lists:map(fun f/1,Xs).
```

After generalisation

```
f(X,Y) -> X+Y.
g(Xs) ->
  lists:map(fun(X)-> f(X,3) end, Ys).
```

This transformation is described by showing how the function *definition* is changed, and also explaining how to change each *use* of the function. We do that thus:

Define the modified function:

```
f(X,Y) -> X+Y.
```

Implement the old function using the new:

```
f = fun(X)-> f(X,3) end
```

How does this describe the refactoring? We can use the implementation to give us the new code, in a series of steps, thus:

```
 g(Xs) -> lists:map(fun f/1,Xs).
-- by substitution giving
 g(Xs) -> lists:map(fun(X)-> f(X,3) end, Ys).
-- after which no rewriting is necessary
```

The implementation of the old function in terms of the new is denoted by `=` rather than `->` to emphasise that this is a semantic equivalence rather than program code defining a function, since the LHS refers to the "old" version of the code and the RHS to the "new".[3]

As earlier, this approach will handle "regular" function applications, in which the `fun` expression will be removed by rewriting, as well as calls to `apply` and `spawn`.

2.3 Other function-oriented examples

Other examples include function unfolding, reordering and regrouping of arguments, adding or removing an argument. We leave it to the reader to verify this. In each case, the general pattern is to present the **new definition** and to describe how to **implement the old function using the new.**

We examine other kinds of refactoring in Section 4.3.

[2] Transforming `fun f/1` into `fun(X)->f(X) end`.

[3] It is a peculiarity of Erlang that the two versions of `f` can co-exist, as functions with the same name but different arity are considered to be distinct.

2.4 Verification

What do we need to establish for the transformed code to be equivalent to the original? The verification factors into two parts:

- For **each specific refactoring** it is necessary to ensure that the “old” function is implemented correctly in terms of the “new”. Specifically, the replacement becomes a *proof obligation*. In the case of renaming, we require that `f` has the same behaviour as
  ```
      fun(X) -> h(X) end
  ```
  when `h` is defined thus:
  ```
      h(X) -> X+1.
  ```
  Similarly for other refactorings.
- On the other hand, **every refactoring** also depends on the correctness of the rewriting rules, such as beta-reduction, eta-expansion, removal of syntactic sugar, etc., which are applied to “tidy up” the resulting code in each case.

3 Towards generic, language-independent refactorings

Refactoring has a very different character in different programming languages; to take one example, [12] compares refactoring in two functional programming languages: Haskell and Erlang. Because of this, the first author was always sceptical about a *language-independent* approach to refactoring. In this Section we argue that our approach of substitution and rewrite allows us to split refactorings into language-independent and language-dependent parts.

At the **language independent** level is a concept like *function application*; function applications can be transformed by *defining the old function in terms of the new*, as described earlier. On the other hand, the **particular** form of function application in different languages varies widely, for example.

- In **Haskell** function applications can be infix `x ’f’ y` as well as prefix `f x y`, and functions – prefix or infix– can also be partially applied, as in the expressions `map (f x) xs` and `map (x ’f’) xs`.
- While **Erlang** does not contain infix function or partial applications, functions are passed as arguments using the “function/arity’ idiom `fun/N`, but also be can be referenced by *atoms* in some special functions, such as `spawn`.

These differences can be dealt with by means of rewriting, as we saw with Erlang earlier. Consider the Haskell example

```
f x y = x+y

g z xs = map (f z) xs
```

where the order of arguments to `f` is reversed, so that the original `f` is implemented in terms of the new thus `\x y -> f y x`. Applying this transformation to the definition of `g` gives

```
g z xs = map (f z) xs
-- substituting the definition of f
g z xs = map ((\x y -> f y x) z) xs
-- by beta reduction
g z xs = map (\y -> f y z) xs
```

This leaves a lambda (‘\’) in the refactored expression, but this is unavoidable. While this example might have been handled better using the `flip` function, this approach generalises to any permutation (or tupling) of the arguments by introducing the appropriate, unnamed, equivalent of `flip` as the lambda expression.

If, on the other hand, we had renamed `f` to `h`, the redefinition of `f` would be `\x y -> h x y`, and the refactoring would completely eliminate the introduced lambda thus:

```
g z xs = map (f z) xs
-- substituting the definition of f
g z xs = map ((\x y -> h x y) z) xs
-- by beta reduction
g z xs = map (\y -> h z y) xs
-- by eta reduction
g z xs = map (h z) xs
```

It is also possible to accommodate infix operations into this framework too. One option is to recognise `’f’` as a function syntax; alternatively, and preferably, we can *pre-process* the code prior to substitution. In this case we proceed thus:

```
g z xs = map (z ’f’) xs
-- replacing the infix "syntactic sugar"
g z xs = map (infix f z) xs
-- substituting the definition of f
g z xs = map (infix (\x y -> h x y) z) xs
-- by eta reduction
g z xs = map (infix (\x -> h x) z) xs
-- by eta reduction
g z xs = map (infix h z) xs
-- reintroducing the infix "syntactic sugar"
g z xs = map (z ‘h‘) xs
```

Language dependence can extend beyond syntactic sugar. For example, in Ocaml function names can appear in signatures and as arguments to functors, where parameters are identified by name rather than position. This impacts the way in which the scope of a renaming refactoring is identified, as explained in [18].

4 Discussion

The approach discussed here is based on some assumptions, and so has some advantages, as well as some limitations. We discuss these in more detail now.

4.1 Local *vs* global

Many refactorings are *local*, in the sense of being applied at a single point, such as a replacement of a double list traversal `map f. map g` by a single one `map (f.g)`, but it is *global* refactorings, whose effect might span multiple sites within multiple modules, that are more problematic to implement and to review, e.g. in a pull request.

We have concentrated on global refactorings here for that reason, but a rewriting approach plainly works well for implementing local refactorings too, as shown by the retrie [17] tool for Haskell. On the other hand, it is difficult to see most local refactorings as anything other than language specific.

4.2 Recursion *vs* iteration

How might the replacement of recursion by iteration be seen in this framework? To encapsulate recursion in general would require some kind of template language, but then an arbitrary recursion cannot be replaced by iteration. Once the recursion has a stylised form, this can be encapsulated in a combinator, as in

```
diffs xs = foldr (-) 0 xs
```

and then a transition to an iterative form can be given thus

```
diffs xs = foldl (flip (-)) 0 (reverse xs)
```

Further transformation can render the list reverse in an iterative way too. This has been expressed in the syntax of Haskell, but all functional languages contain cognates of lists and folding operations, and so, arguably, it has a language-independent aspect.

4.3 Other kinds of refactoring

A similar approach can be taken to *constructor-based* refactorings in languages like Haskell and OCaml. A constructor is like a function, except that it can be used on the “left hand side’ of definitions in pattern matches, and this requires some limited form of rewriting on patterns to implement.

There are limits to the approach described here. For example, ‘folding’ function definitions, i.e. replacing instances of a function body by a call to the function necessitate replacing (an instance of) a complex expression, rather than a single term. In the short term, we aim more clearly to articulate the scope and limits of the approach.

5 Related work

5.1 Language-independence and genericity

The questions of language-independence and genericity for refactorings have been addressed before. Indeed, Eelco Visser initiated a discussion of this in the millennium year in *Language Independent Traversals for Program Transformation* [25], which described how strategic, traversal-based programming could achieve transformations such as change in bound variable names across functional and OO languages that could be subsets of Haskell and C++ respectively. Shortly after this Lämmel’s *Towards Generic Refactoring* [11] took this explicitly to the example of function/abstraction extraction with an approach that performs a conceptual analysis of the categories of transformations and pre-conditions that are necessary for a generic treatment of a refactoring.

This approach is flexible and comprehensive, but it lacks completeness. While it can encompass the generic features that occur in multiple languages, and indeed adapt e.g. to the transition between expression- and statement-oriented languages, it does not support the particularities of different languages, such as operator sections in Haskell or the use of atoms to denote functions in Erlang, that our approach can handle.

While we have argued that our approach supports a degree of language independence, we would not claim that it directly supports multi-language refactoring [21], since that requires not only awareness of the separate semantics of a number of languages, but also their semantic interactions.

5.2 Verification of refactorings

A powerful approach to ensuring the correctness of refactorings is to ensure that they meet the set of constraints that embody (aspects of) the semantics of the programming language being refactored. This insight was first presented by Tip and colleagues in the context of preserving *type constraints* [24], and elaborated for Java by de Moor and Schaefer [20]. Steimann [22] presents a general theory of constraint-based refactoring, and outlines a program in which correct-by-construction tools can be built on top of constraint-based presentations of programming language semantics.

Pioneering work by Kniesel and Koch [10] examines the way that correct refactorings can be built by composing simpler parts that themselves preserve behaviour, or can be verified separately. Our approach is related, but differs in that different instances of the same refactoring will involve different rewrites, depending on the context of the instance: the composition is thus, in a sense, dynamic.

5.3 API migration

When an API is upgraded it can be taken for granted that the new API should afford all the functionality provided by the old version; this can be made concrete in an *adapter module* that defines the old in terms of the new.

While adaptation is enough to ensure that the client system continues to work, it has disadvantages. If an API evolves continually, then a series of adapter modules will stack up, and even in the case of a single adaptation, the code will be neither idiomatic nor natural. One approach to this is to generate transformations from the replacement code [16], which ensures that the explicit wrapper code disappears. This mechanism is extended by our approach, outlined in [13], where the replacement code is subsequently simplified by rewriting, e.g. removing case expressions when they can be resolved, or exception-handling code when that is unnecessary.

This adaptation can be complex, however, particularly in the case of object-oriented programming, and especially when the migration is from one API to another, unrelated one. Lämmel and colleagues outline this in a case study [1] of evolving a system, while providing a broad overview of previous approaches, as well as in this general exploration of two XML case studies here [2].

It turns out that the approach we outline in this paper can be seen as a particular case of the work presented in [13], viewing each refactoring as an evolution of an API for those aspects of the code that has changed, and also illustrated in this work on API migration in OCaml [6].

5.4 Refactoring schemes

The approach explained in this paper can be seen as a variant of the *refactoring schemes* proposed in [7]. In particular, the examples given in Section 2 instantiate the function refactoring scheme, which can be understood as a strategy that changes function entities in a program by applying rewrite rules to the definition and to the references of the function. When restricting the rewrite rules to only rewrite the name of the function in the reference, the rewrite step does not perform pattern matching and thus it becomes a simple substitution.

6 Conclusions and future work

We have advanced an argument that it is possible to view general refactorings as having a language-independent component, described in the language of function application, and a language-dependent component, materialised by a set of language-specific rewrite rules. This description re-frames earlier work of ours on refactoring for API evolution and language schemes.

We have experimental implementations of the general function refactoring introduced in Section 2 in the Wrangler [14] refactoring tool for Erlang, where it is materialised as an Erlang `behaviour`, and in the Rotor [19] refactoring tool for OCaml. It is a short term goal to finalise and deploy these implementations, as well as articulating the scope and limits of the approach itself.

This work forms part of a longer-term project to build high assurance refactorings. Earlier work on this has concentrated on a formal treatment of (re-)naming in OCaml [18], and a formalisation of the semantics of Erlang [3].

We are very grateful to the referees for their feedback, and in particular their encouragement to contextualise the work more thoroughly, as well as to the ROTOR team at Kent for their insights into refactoring OCaml programs.

References

1 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In Radu Marinescu, Michele Lanza, and Andrian Marcus, editors, *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010. `doi:10.1109/ICSM.2010.5610429`.

2 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs Van Der Storm. Study of an API migration for two XML APIs. In *International Conference on Software Language Engineering*, pages 42–61. Springer, 2010.

3 Péter Bereczky, Dániel Horpácsi, and Simon Thompson. A Proof Assistant Based Formalisation of a Subset of Sequential Core Erlang. In Aleksander Byrski and John Hughes, editors, *Trends in Functional Programming*, pages 139–158, Cham, 2020. Springer International Publishing.

4 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72, 2008.

5 Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. QuickChecking Refactoring Tools. In Scott Lystig Fritchie and Konstantinos Sagonas, editors, *Erlang'10: Proceedings of the 2010 ACM SIGPLAN Erlang Workshop*, pages 75–80. ACM SIGPLAN, 2010.

6 Joseph Harrison, Simon Thompson, Steven Varoumas, and Reuben Rowe. API migration: `compare` transformed. In *OCaml Workshop 2020*, 2020.

7 Dániel Horpácsi, Judit Kőszegi, and Zoltán Horváth. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. *Electronic Proceedings in Theoretical Computer Science*, 253:92–108, August 2017. `doi:10.4204/eptcs.253.8`.

8 Marie-Christine Jakobs. PEQCHECK: Localized and Context-aware Checking of Functional Equivalence. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 130–140, 2021. `doi:10.1109/FormaliSE52586.2021.00019`.

9 Marie-Christine Jakobs and Maik Wiesner. PEQtest: Testing Functional Equivalence. In Einar Broch Johnsen and Manuel Wimmer, editors, *Fundamental Approaches to Software Engineering*, pages 184–204, Cham, 2022. Springer International Publishing.

10 Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1):9–51, 2004. Special Issue on Program Transformation. `doi:doi:10.1016/j.scico.2004.03.002`.

11 Ralf Lämmel. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming*, RULE '02, pages 15–28, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/570186.570188.

12 Huiqing Li and Simon Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In M. Di Penta and L. Moonen, editors, *Source Code Analysis and Manipulation, SCAM'06*, 2006.

13 Huiqing Li and Simon Thompson. Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs. In Tim Menzies and Motoshi Saeki, editors, *Automated Software Engineering, ASE'12*. IEEE Computer Society, 2012.

14 Huiqing Li, Simon Thompson, György Orosz, and Melinda Töth. Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada*, 2008.

15 Ralf Lämmel and Joost Visser. A Strafunski Application Letter. *Information and Computation/information and Control - IANDC*, pages 357–375, January 2003. doi:10.1007/3-540-36388-2_24.

16 Jeff H. Perkins. Automatically Generating Refactorings to Support API Evolution. *SIGSOFT Softw. Eng. Notes*, 31(1):111–114, September 2005. doi:10.1145/1108768.1108818.

17 Retrie, a powerful, easy-to-use codemodding tool for Haskell., 2020. URL: https://github.com/facebookincubator/retrie.

18 Reuben Rowe, Hugo Férée, Simon Thompson, and Scott Owens. Characterising renaming within Ocaml's module system: theory and implementation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 950–965, 2019.

19 Reuben Rowe, Hugo Férée, Simon Thompson, and Scott Owens. ROTOR: A Tool for Renaming Values in OCaml's Module System. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 27–30, 2019. doi:10.1109/IWoR.2019.00013.

20 Max Schaefer and Oege de Moor. Specifying and Implementing Refactorings. *SIGPLAN Not.*, 45(10):286–301, October 2010. doi:10.1145/1932682.1869485.

21 Hagen Schink, Martin Kuhlemann, Gunter Saake, and Ralf Lämmel. Hurdles in Multi-language Refactoring of Hibernate Applications. In *ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Database Technologies*, volume 2, pages 129–134, January 2011.

22 Friedrich Steimann. Constraint-Based Refactoring. *ACM Trans. Program. Lang. Syst.*, 40(1), January 2018. doi:10.1145/3156016.

23 Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation*. ACM SIGPLAN, 2008.

24 Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring Using Type Constraints. *ACM Trans. Program. Lang. Syst.*, 33(3), May 2011. doi:10.1145/1961204.1961205.

25 Eelco Visser. Language Independent Traversals for Program Transformation. In J. Jeuring, editor, *Proceedings of the Workshop on Generic Programming (WGP2000), Ponte de Lima, Portugal*, July 2000. Technical Report, Department of Information and Computing Sciences, Universiteit Utrecht.

Eelco Visser: The Oregon Connection

Andrew Tolmach
Portland State University, OR, USA

Abstract

This paper shares some memories of Eelco gathered over the past 25 years as a colleague and friend, and reflects on the nature of modern international collaborations.

2012 ACM Subject Classification Social and professional topics → Historical people

Keywords and phrases Eelco Visser, Stratego, scope graphs

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.27

1 OGI and Stratego

I first met Eelco in June 1997 on the campus of the University of Amsterdam, which was hosting the International Conference on Functional Programming (ICFP). Eelco was just finishing up his Ph.D. on parsing, and had been offered a postdoc position at the Oregon Graduate Institute (OGI) in Portland, where I held a joint appointment in what we called the Pacific Software Research Center (PacSoft). I believe the postdoc invitation initially derived from a connection between Eelco's thesis advisor, Paul Klint, and PacSoft's senior faculty member, Richard Kieburtz. But as someone who worked on practical aspects of (functional) language compilation and tools, I seemed like the faculty member whose interests best matched Eelco's, so it fell mainly to me to sell him on the idea of coming to Oregon. This was not so easy: Eelco already had a clear idea of his career path, and he wanted to be sure that his postdoc year would help him advance along it, and not just be a pleasant vacation after completing his degree. Fortunately, I was able to convince him that PacSoft, which was focused on methods for design and development of domain-specific languages (DSLs), would be a good environment in which to pursue his interests in language tooling.[1]

When Eelco arrived in Portland that Fall, he was already engaged with the idea of developing a particular (meta) DSL of his own, namely a language for defining term rewriting strategies separately from the underlying rewrite rules, using combinators somewhat inspired by operators in process calculi. This idea was motivated by his experiences using existing rewriting tools such as ASF+SDF and ELAN, and it had already been the subject of an initial paper with Bas Luttik [3]. At PacSoft he built several prototype implementations of the language, which shortly after acquired the name Stratego. The first implementation was in MetaML [7], which was being developed by Tim Sheard and Walid Taha at OGI at that time; this was followed by a boot-strapped implementation that ultimately generated C code using the ATerm library [11]. Eelco also worked with another postdoc, Zino Benaissa, to develop an elegant core theory for the strategy combinators [14]. The three of us collaborated to write about the language in the context of a particular application: optimization of expressions in a compiler for RML [8], a dialect of ML developed a little earlier at PacSoft by Dino Oliva and myself. The resulting paper, "Building Program Optimizers with Rewriting Strategies," appeared at ICFP98 [15], and became a foundational citation for Stratego and strategic programming in general.[2]

[1] Given Eelco's life-long love of photography, I suspect that Oregon's (well-deserved) reputation for natural beauty probably helped too.

[2] Incidentally, according to a friend on the program committee, the paper very nearly wasn't accepted. As of 2012, it was the fifth most highly cited paper in the history of the ICFP conference.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 27; pp. 27:1–27:6
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Eelco was a great postdoc, of course, because of his ability, energy, and congenial personality. We used to enjoy long rambles around the (rather bucolic) OGI campus; we both found walking conducive to creative thinking out loud. But what impressed me the most about Eelco was his willingness and ability to sit down and *write*[3] about his research ideas while they were still in their formative stages. This quality is pretty rare among aspiring computer scientists, and may be a good predictor of success. In later years, Eelco was kind enough to blame me for having impressed upon him the importance of regular conference publication ("got me going on the treadmill" is the phrase he used). But he certainly didn't need me to teach him that writing helps to clarify thinking, and that research achievements are useless until they are effectively communicated to others.

Stratego became the foundation of Eelco's entire career, both literally and figuratively: it was a key component of nearly all the software systems he built, and it served as his distinctive calling card in the research community. This has perhaps been a mixed blessing: the user base for the language has remained small, which is a barrier to entry for those wishing to contribute to (or in some cases, even to use) Stratego-based tools. On the other hand, using Stratego does encourage novel techniques and perspectives that are not so natural in more type-centered functional languages such as Haskell or ML.[4] Indeed, my experience when observing Eelco program was that he approached most tasks as *syntactic* transformation problems; he always saw the ATerms lurking not far below the surface.

2 Verification and Scope Graphs

At the conclusion of his postdoc, Eelco returned to the Netherlands to take up a faculty position at Utrecht. We exchanged a few visits over the next couple of years, in Portland and in Amsterdam. After that, we met only occasionally at conferences during the subsequent decade. The Stratego mailing list provided a periodic update on his many activities. I was very impressed by the high productivity of Eelco's research group, which he was quick to credit to having excellent students – although he did take pride in his ability to attract and keep them.

Our second period of active collaboration began in 2011. By this time, Eelco had moved to Delft, built a new group, and completed many parts of the Stratego-based language workbench that was now called Spoofax. He was becoming interested in adding a verification dimension to the workbench tools, for example to automatically generate proofs of type soundness for a language definition. Knowing that my own interests had shifted toward verification, he invited me to join in a new research effort to extend Spoofax in this direction [16], which he was able to support with a large (and prestigious) Dutch VICI grant. It is quite pleasing when a former student or postdoc thinks you have something to contribute to their (already) successful enterprise!

As it turned out, the first major fruit of this collaboration was on a somewhat peripheral topic: formalizing the notion of name binding in languages. Spoofax already included a tool called NaBL for specifying name binding, but the NaBL language lacked a clear semantics independent of its current concrete implementation. Since binding resolution is a key requirement for defining static typing, it seemed reasonable to attack this problem before

[3] And often *draw*. His art background evidently encouraged him to express ideas visually too.

[4] In the early days of Stratego, we often used to argue about the costs and benefits of static typing. Despite the early work of Ralf Lämmel [4], it is only quite recently that Stratego has acquired a (gradual) type system as part of the ongoing Ph.D. work of Jeff Smits [6].

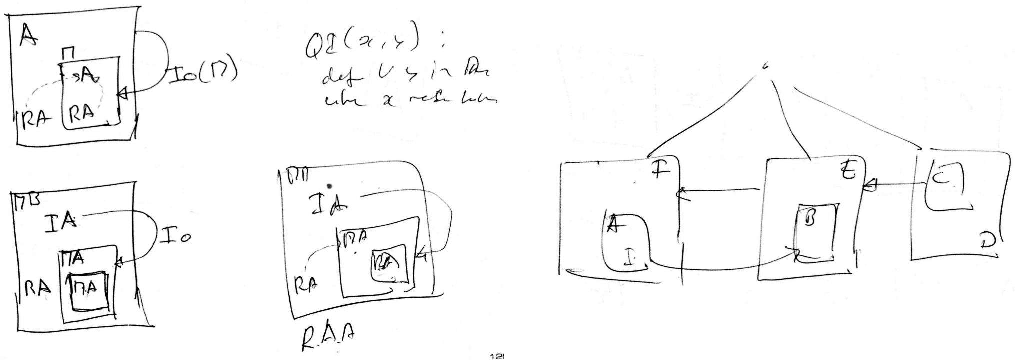


Figure 1 Some of Eelco's highly caffeinated initial sketches for scope graphs.

trying to address verification of type soundness. This effort resulted in the invention of *scope graphs*, a simple but powerful formalism for describing binding structure, with a strong visual intuition. The idea first really took concrete shape under Eelco's pen in a Portland coffee shop in June 2014 (see Figure 1). The paper introducing scope graphs [5], joint with Delft postdoc Pierre Néron and faculty member Guido Wachsmuth, won the Best Paper award at the European Symposium on Programming (ESOP) conference in 2015. We were quite proud of this paper, especially its balance of theory and practicality.

The initial work on scope graphs spawned a good deal of subsequent research, some of which I joined, together with several Delft students, postdocs, and faculty, including Hendrik van Antwerpen, Casper Bach Poulsen, Vlad Vergu, Arjen Rouvoet, and Robbert Krebbers. A first effort was to use scope graphs to support type checking, which clarified that binding and typing are not wholly separable concerns, and led to a unified constraint-based approach to defining and querying scope graphs [10].[5] We also eventually returned to our original verification goals via an approach to dynamic semantics based on heap *frames* that derive their shape and connections from an underlying scope graph. Scopes-and-frames has proved a fruitful paradigm for simplifying proofs of soundness [1], structuring intrinsically typed interpreters [2], and supporting optimization [12]. And there is still more being mined from the basic idea of scope graphs.

Scope graphs are largely about describing data that is explicitly named by the user. But we gradually came to realize the importance of also handling other dynamic data, e.g. the temporaries and control information typically produced by a compiler. This led to a number of lower-level models for defining dynamic semantics, under the name Dynamix, which were an active topic of collaboration among Eelco, Casper Bach Poulsen, and myself during the pandemic years. Much of the work was planned in the form of projects conducted by Delft master's students Chiel Bruin, Bram Crielaard, Thijs Molendijk, and Ruben van Baarle. Eelco's death occurred before the latter two had completed their degrees; Casper and I have done our best to supervise them since then.

The scope graphs collaboration was extremely rewarding, if sometimes a bit intense. Eelco had a persistent habit of juggling far too many tasks at once, and scope graphs were just one cog in his busy research machine. So even without conscious procrastination, lots of things didn't get timely attention, and there would inevitably be a mad drive to complete

[5] Subsequent work, with which I was not directly involved, extended these ideas to deal with structural typing and led to the design of the Statix DSL for static semantics [9].

the work backing a conference submission in the last few days before a deadline. I think Eelco thrived under this regimen. Already in the Preface to his Ph.D. thesis [13, p. iv], he had rhapsodized

> Finally, one of the great contributors to this thesis is the deadline. Soft deadlines, firm deadlines, extended deadlines, nearing deadlines, changed deadlines, passed deadlines, the empowering deadlines that make page after page appear as if by magic, the deadlines that spook at night, deadlines that you can really feel, deadlines that are suddenly there...

While I still prefer to keep my deadlines at a distance, I discovered to my surprise that quite a bit of good science could be done under this kind of time pressure (writing clarifies thinking, once again).

3 Talking to Delft

While Eelco and I did visit each other fairly frequently (thanks in large part to that VICI grant), much of our collaboration was conducted at a distance. Over most of the last ten years, we talked face to face every week or two, courtesy of Skype or Zoom. I still find this quite amazing just on a technological level. When I started my career, even domestic long-distance voice conversations were still expensive, and calls between Europe and the US were only for emergencies. Now it is possible to have a visual, interactive research discussion at an arbitrary distance, essentially for free. It is hard to overestimate how much this development aids international collaboration.

One barrier that does remain is the time difference. Delft is nine hours ahead of Portland, so finding a mutually acceptable time to meet with Eelco required compromise: our regular meeting slot was a little early for me and a little late for him. Paradoxically, I think this slight inconvenience helped us take the meetings more seriously: once they were on the schedule, we didn't want to cancel them lightly.

Our conversations usually focused on the research problem of the moment, and typically included other postdocs and students. But sometimes it was just the two of us, and sometimes we were just checking in to find out how the other was coping with the latest professional, political, or pandemic news. On the professional front, it was an opportunity to speak freely – brag a little or complain a lot – in ways that would be difficult with colleagues in our home institutions. On a personal level, it was a chance to to share ideas across different nations and cultures;[6] Eelco was always interested and engaged in the world beyond computer science. I've spent quite a few happy years living and working in Europe, and talking regularly with Eelco was a way of keeping that transatlantic connection alive – which seems more important than ever in our increasingly separatist world.

In short, my regular talks with Eelco were a rare privilege. I sorely miss them. I sorely miss him.

[6] It could also have been a chance to share them in different languages, except that I have never learned any Dutch at all, to my regret.

References

1 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.20.

2 Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158104.

3 Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

4 Ralf Lämmel. Typed generic traversal with term rewriting strategies. *The Journal of Logic and Algebraic Programming*, 54(1):1–64, 2003. doi:10.1016/S1567-8326(02)00028-0.

5 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

6 Jeff Smits and Eelco Visser. Gradually typing strategies. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 1–15. ACM, 2020. doi:10.1145/3426425.3426928.

7 Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211–242, 2000. PEPM'97. doi:10.1016/S0304-3975(00)00053-0.

8 Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Program.*, 8(4):367–412, 1998. doi:10.1017/s0956796898003086.

9 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.

10 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

11 Mark van den Brand, Hayco de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30, 2000.

12 Vlad A. Vergu, Andrew P. Tolmach, and Eelco Visser. Scopes and frames improve meta-interpreter specialization. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.4.

13 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

14 Eelco Visser and Zine-El-Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, 1998. doi:10.1016/S1571-0661(05)80027-1.

15 Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26, Baltimore, Maryland, United States, 1998. ACM. doi:10.1145/289423.289425.

16 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. doi:10.1145/2661136.2661149.

Eelco Visser and IFIP WG 2.16

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
University of Groningen, The Netherlands

Abstract

Eelco Visser was a founding member of the IFIP TC2 Working Group 2.16 and long served as its chair. This brief note recounts Eelco's impact on the group and his contributions to its meetings[1].

2012 ACM Subject Classification Software and its engineering → Formal language definitions; General and reference → Biographies

Keywords and phrases Eelco Visser, Language design, IFIP, working group

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.28

1 Introduction

Eelco Visser in Austin, 2012. Photographs by William Cook†.

2 Lunch in Reno

The IFIP TC2 Working Group 2.16[2] was conceived at an informal lunch at SPLASH 2010 in Reno, NV, colloquially nicknamed the "minus 1st" meeting. A number of researchers, including Eelco, had observed a lack of space to discuss programming language design in the broad sense. Indeed, the mission statement of the group states:

> *We have noticed that researchers interested in language design are isolated and lack a place to exchange ideas and criticism. Computer science conferences no longer serve this role, because they have become fixated on rigorous evaluation. There is nothing*

[1] An early draft was circulated among the membership of WG 2.16 for comments.
[2] `http://languagedesign.org`

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 28; pp. 28:1–28:3
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

> *wrong with rigorous evaluation of things that can be so evaluated, but language design ideas, particularly in their formative stages where feedback is most crucial – when there may be neither implementation nor experience with their use – cannot be so evaluated.*

The result of this meeting was an official application to IFIP TC2, chaired by Bertrand Meyer at the time, presented by Jonathan Edwards and Eelco. The application was successful, and WG 2.16 was born.

The inaugural meeting was held the next year, in Mountain View, CA. And since then Eelco attended almost every meeting. Only when travel restrictions and lockdowns forced us to organize the meeting online he started skipping sessions, or did not attend the meeting altogether. In 2023, we are organizing our first physical meeting since the peak of the pandemic. Sadly, it will be without him.

3 Eelco's contributions to the meetings

Eelco was a prolific researcher, but also a great designer, with a keen eye for detail, and attention for good taste. His slides were often beautifully designed. And when he decided to give a talk without slides at all, it was a carefully prepared live coding session, showing off his language engineering tools.

His talks often started with "Soooooo, I've been working on x" where $x \in \{$name binding, web programming, ...$\}$. The following list gives a good overview of the topics he covered at the working group meetings:

- Exploring the Web Programming Design Space (Mountain View, 2011)
- Dimensions of Domain-Specific Language Design (Mountain View, 2011)
- Spoofax Language Workbench (London, 2012)
- Declarative Name Binding and Scope Rules (Austin, 2012)
- Linguistic Abstractions for Web programming (Aarhus, 2013)
- Dynamic Semantics Specification in DynSem (Athens, 2015)
- The Name Binding Game (Los Angeles, 2016)
- The Semantics of Name Resolution in Grace (Park City, 2017)
- The Syntax Definition Formalism SDF3 (Antwerp, 2018)

Eelco was also an active ambassador for his work, always on the look-out for potential users of his tools or opportunities for collaboration. "Let's meet during the break, and I'll show you.", he'd say, almost coercively, and nobody dared to resist.

In a sense, Eelco was the ideal chairperson, since he hovered above (or in between) the factions, styles, and schools of the programming language design field. Thematically, he occupied a middle ground between "the formal ones" (type systems, semantics, verification, etc.) and the "free radicals" (live programming, new programming interfaces, etc.).

If you look at the list above, however, you immediately observe that he was a "meta" person: most of his talks related to his ongoing work on the Spoofax language workbench and its set of meta-languages (SDF3, DynSem, NaBL, Statix). The exception being WebDSL[3], his language for web programming, implemented in Spoofax[4].

He once lamented to me in private, that "we need more language engineering stuff". What he meant by that is: we could improve language *design* by developing better tools for language *engineering*. I think this summarizes his long-term goals quite accurately: better tools free up mental space for better design.

[3] https://webdsl.org/

[4] https://spoofax.dev/

4 Chairing WG 2.16

Since the beginning, Eelco served as chair of the working group, until I succeeded him in 2019. In this role, he was in charge of scheduling talks during the meeting. The way he did this was: he'd walk around behind the attendees, sitting in carré arrangement, then squatted down and looked at you with his characteristic twinkling in his eyes: "So you will talk about x, right?", or "Can you be the first speaker tomorrow?" (smirk).

His most direct impact for me personally is that he invited me as a visitor of the working group in 2012. And together with William Cook, who was vice-chair at the time, he asked me to become a member at the Austin meeting in 2012.

Everyone who knows Eelco, has seen him walking around with his camera. At the working group he acted as our semi-official photographer. Most of his working group pictures can be found online at `https://www.flickr.com/photos/eelcovisser/albums/72157629257337544`.

5 In memoriam

Eelco's untimely and sudden death happened less than a year from the passing of another founding member: William Cook (†2021). WG 2.16 is still struck by these events. Both persons have been inspiring to me, both personallly and professionally. I am sure the working group will keep their memory alive.

Only recently I learned that the official title of IFIP Working Group 2.16 is "Programming Language Design". Turns out that Eelco had preferred "Language Design" (probably to include DSLs, specification languages, and meta languages), and that became the de facto name. I think this is exemplary for how he ran the working group: subtle, but impactful.

Semantics Engineering with Concrete Syntax

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
University of Groningen, The Netherlands

Abstract

Semantics engineering tools like Redex can be used to define, explore, and debug formal definitions of programming language semantics. However, such tools are often based on abstract syntax, which makes the definition of rules and the exploration of execution traces rather unfriendly. In this paper we introduce CREDEX, a library in the Rascal meta-programming language for defining small-step evaluation-context semantics, where terms and matching patterns are what-you-see-is-what-you-get. CREDEX employs parsing for decomposing terms into context and redex. Since Rascal's grammar formalism is based on general parsing, a non-unique decomposition of a term literally corresponds to an ambiguous parse. We demonstrate the use of CREDEX, detail some aspects of its implementation, and discuss three case-studies.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Integrated and visual development environments

Keywords and phrases Semantics engineering, syntax, parsing, reduction semantics

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.29

1 Introduction

Operational semantics with evaluation contexts [10] is a popular framework to define the semantics of programming languages using rewriting rules. The use of contexts to separate traversal from the actual reduction rules leads to concise and modular definitions.

Specifications of formal semantics come to life by making them executable. It has been observed that semantic specifications could form central artifacts in language engineering [11]. Tools such as as Redex [9, 18] and K [23] allow language designers to explore, inspect, test, and debug their language designs. Redex, for instance, has been instrumental to uncover numerous bugs in published formal definitions [13]. Both K and Redex have been used to formalize realistic (subsets of) programming languages (e.g., [5, 17]).

Language workbenches [7, 8] such as Rascal [16] and Spoofax [12] offer integrated meta-formalisms and IDE services dedicated to the principled design and implementation of software languages. Nevertheless, most of the existing language workbenches lack support for semantics engineering. Although the ASF+SDF Meta-Environment was based on algebraic specification and term rewriting, and both Spoofax and Rascal inherited those features, this means that the default style of defining a semantics is through definitional interpreters, which are limited to big-step evaluation, and do not facilitate the exploration and debugging offered by, e.g., Redex. The recent work on DYNSEM [26] aims to bridge the gap between interpreters and formal semantics, but still stays close to natural, big-step semantics.

In this paper we present CREDEX, a framework for defining executable semantics, in the small-step style of Redex. CREDEX's main selling points are: it is based on *concrete* syntax, rather than abstract syntax (e.g., prefix notation, or s-expressions), and, because CREDEX is a library in Rascal, it integrates very well with the other language workbench features of Rascal. CREDEX is novel in its (rather unconventional) use of parsing and (ambiguous) parse forests to drive the reduction process.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 29; pp. 29:1–29:11
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Eelco Visser has always been a syntax person. His PhD thesis centered around syntax definition using SDF. He published many papers about concrete syntax embedding, parsing, and syntax definition after that. Recently, he had ventured into dynamic semantics as well. C*REDEX combines all of these strands of research.*

2 Overview

Structural operational semantics in the style of Plotkin [21, 22] is based on rewriting (or reduction) of syntactic terms, where each rewrite step corresponds to a small-step execution step. The execution of a term is finished when it reduces to a value or no further rules apply (the term is "stuck"). A common slogan heard in this context is "everything is syntax", because both results (values) and auxiliar entities (stores, environments) are represented syntactically.

Ordinary operational semantics involves defining reduction rules that merely traverse the term until a redex is found "where something happens". Operational semantics with evaluation contexts (also known as context-sensitive reduction semantics) avoids this boilerplate by defining a context grammar, which captures the traversal through a term separately. For a syntactic sort e, a context grammar could look like this:

$$\begin{array}{lll} e & ::= & e * e \mid e - e \mid n \\ E & ::= & \square \mid E[\square * e] \mid E[e * \square] \mid E[\square - e] \mid E[n - \square] \end{array}$$

The nonterminal E defines patterns of terms where the box $\square$ represents a placeholder for a term to be plugged in. In this case the contexts define out-of-order evaluation for multiplication, and left-to-right evaluation for subtraction. Operational semantics with evaluation contexts works by first *splitting* a term into a context $E[\ ]$ and a *redex* (ocurring at the placeholder position). When a redex has been reduced, it is *plugged* back into the context and execution continues. The reduction rules only need to be defined for the redexes, thus avoiding a lot of boilerplate.

Tools like Redex implement this process of splitting/plugging on top of abstract syntax definitions. As a result, both the specifications and visualization tools employ s-expression notation to write and display terms. Systems like Rascal and Spoofax support meta-programming with *concrete syntax* (as pioneered by ASF+SDF [15]), which has some distinct benefits:

- Pattern matching with concrete syntax is WYSIWYG: object-level terms in the meta-program read like literal object language expressions.
- If terms are internally represented as concrete syntax trees, then rendering a term to object-language syntax is literally for free; no need for pretty printers.
- Concrete syntax trees can be transformed while maintaining layout information, thus making the display of intermediate and final results more human-friendly.
- If a concrete syntax grammar is already available to obtain a parser for a language, why not reuse it for the definition of a semantics?

CREDEX takes these considerations to heart: it starts with the actual concrete syntax grammar (as defined in Rascal), and then allows it to be modularly extended with a *concrete context grammar*. This second grammar drives the splitting process through the derived parser. Although the link between context-redex decomposition and context-free grammars has been made before [27], as far as the author is aware, CREDEX is the first tool to actually perform decomposition through parsing, and to leverage the potentially ambiguous parse forest for representing non-unique decompositions. Below we introduce CREDEX using an example.

```
syntax Expr
  = Num
  | bracket "(" Expr ")"
  | assoc Expr "*" Expr
  > left Expr "-" Expr
  ;

lexical Num
  = [\-]?[1-9][0-9]* | [0] ;
```

```
syntax E
  = "(" E ")"
  | E "*" Expr
  | Expr "*" E
  | E "-" Expr
  | Num "-" E
  | @redex "(" Num ")"
  | @redex Num "*" Num
  | @redex Num "-" Num;
```

Figure 1 Specifying concrete syntax (left) and concrete evaluation contexts (right).

3 Credex by example

3.1 Defining contexts

The left-hand side of Figure 1 shows a simple expression grammar in Rascal's built-in grammar formalism. The definition of whitespace is omitted, but it is implicitly included inbetween the symbols of a `syntax` declaration. An `Expr` is defined as a literal number, a bracketed expression, multiplication (which is `assoc`iative), and subtraction (which has lower precedence than multiplication due to the use of `>` instead of `|`). The last line of the grammar defines the lexical syntax of numbers using character classes and regular expressions.

The right-hand side of the figure shows a context grammar for expression contexts, which are conventionally named `E`. The way to read such a grammar is by viewing the occurrences of `E` as directions for tree traversal. So, in the first alternative, the only way to go, is inside the parentheses. For multiplication, the traversal can proceed either down the left-hand side or down the right-hand side, as indicated by two productions having `E` at the left-hand side, or at the right-hand side, respectively. Note that this is a language design decision: we are *defining* here that the evaluation order of multiplication is arbitrary. For subtraction, however, the two productions are asymmetric: only after having evaluated the left-hand side to a number (`Num`) can evaluation continue in the right-hand side. This enforces strict left-to-right evaluation for subtraction.

Finally, the grammar includes three productions annotated with `@redex`, to indicate sub-terms that are "interesting", and can be reduced are part of a step. In other words, these productions describe the sub-trees where the traversal via the `E` nonterminal should stop. Note that the context grammar (trivially) does not generate the original language, because there is no `E`-production to generate a single `Num`, which is a valid expression. This makes intuitive sense, however, since numbers are irreduceable.

3.2 Splitting Through Parsing

Splitting in Credex works as follows: the process starts with a (parsed, non-ambiguous) term over the object language (e.g., of type `Expr`). The term is then unparsed to text, and parsed *again*, but this time over the context grammar (e.g., `E`). This returns a (possibly ambiguous) parse forest, where certain sub-nodes are annotated with `@redex`. Split analyzes the parse forest, and produces a list of pairs corresponding to the contexts and redexes. The context is a parse-tree where the redex sub-node has been replaced with a designated place holder "□". Every ambiguity in the forest adds another context-redex pair.

Let's say we have the term `(1 - 2) * (2 - 3)`. Calling split on this term gives us a non-unique decomposition: 1) `(□) * (2 - 3)`, with redex `1 - 2`, or 2) `(1 - 2) * (□)`, with redex `2 - 3`. This corresponds to the definition of the contexts for multiplication: it's allowed to first evaluate the left-hand side, or the right-hand side.

```
CR red("par", E e, (E)`(<Num n>)`) = {<e, (Expr)`<Num n>`>};
CR red("mul", E e, (E)`<Num n1> * <Num n2>`) = {<e, [Expr]"<toInt(n1) * toInt(n2)>">};
CR red("sub", E e, (E)`<Num n1> - <Num n2>`) = {<e, [Expr]"<toInt(n1) - toInt(n2)>">};
default CR red(str _, E _, Tree _) = {}; // we're stuck or done.

RR applyExpr(Expr e) = apply(#E, #Expr, red, e, {"mul", "sub", "par"});
```

Figure 2 Reduction rules for the expression language.

Changing the example term to `(1 - 2) - (2 - 3)`, however, produces just a single decomposition: `(□) - (2 - 3)`, with redex `1 - 2`, because, according to the context grammar, evaluating subtraction needs to start at the left-hand side. Things are not thát simple, however, because of accidental ambiguities in the context grammar that we are unaware of, but we will address this problem in Section 3.4. Let's first look at how to specify reduction rules.

3.3 Specifying Reduction Rules

Reduction rules are specified as a case-based Rascal function, conventionally named `red` (short for "reduce"). The rules for the expression language are shown in Figure 2. Note that Rascal functions dispatch based on the patterns of their arguments, with the benefit that Credex specifications are modularly extensible by simpling adding additional cases for new combinations of syntax.

The first argument of `red` matches on is a literal string constant, acting as a rule label. The second argument is the context (in this case of type `E`). The third argument employs concrete syntax matching on `E`-contexts, where the part between backticks captures the redex. Note that this pattern is not a string, but tree pattern expressed in the concrete grammar of the object language, where fish-angle brackets are used to introduce (typed) pattern variables, such as `<Num n>`.

The result type of `red`, is `CR` a binary relation type associating a (possibly) modified context to a reduct[1]. Each case of `red` matches on a rule label (e.g., `mul`), the input context, and the redex. Observe that the rules employ concrete syntax matching (the parts between backticks) as defined by the context grammar, and note that the rules only match on syntax annotated with `@redex`. The `mul` and `sub` rules both use the `[Expr]` parsing-operator to create new expressions from the result of multiplication and subtraction, respectively.

The helper function `applyExpr` is used to have a term perform a single step, according to the set of rules identified by the set of labels provided to the generic function `apply`. The `apply` (Figure 3) function uses the reified types (`#E` and `#Expr`) to perform splitting[2]. It then iterates over the set of rule labels, tries invoking the `red` function, and unions the result(s). The result type, `RR`, is a relation from rule-label to terms, capturing which steps the input term `e` could have performed and with what result.

3.4 Eliminating Spurious Ambiguities

Let's return to splitting a term by parsing using the context grammars. In the above examples we got precisely the decompositions that we had wantend: two for the multiplication, and one for subtraction. If we consider the term `1 - 2 - 3`, however, a naïve splitting according to the grammar of Figure 1, will result in two decompositions: □ `- 3` with redex `1 - 2`, and `1 -` □, with redex `2 - 3`, which is incorrect.

[1] In a sense we use the relation type as an option type. We could have used Rascal's `Maybe` type, but the curly braces incur less syntactic noise than `just`-constructors.

[2] Reified types are Rascal's reflection system; think of `#E` as similar to `E.class` in Java.

```
rel[str,Tree] apply(type[Tree] C, type[Tree] T, CR(str,Tree,Tree) red, Tree t, set[str] rules) {
  result = {};
  for (<ctx, rx> ← split(C, T, t)) {
    for(l ← rules) {
      for (<ctx2, rt> ← red(l, ctx, rx)) {
        result += {<l, plug(ctx2, rt)>};
      }
    }
  }
  return result;
}
```

Figure 3 Pseudocode for `apply`.

```
  syntax E
    = "⟨" E "*" Expr "⟩"
    | "⟨" "(" E ")" "⟩"
    | "⟨" Expr "*" E "⟩"
    | "⟨" Num "-" E "⟩"
    | "⟨" E "-" Expr "⟩"
    | @redex "⟨" Num "-" Num "⟩"
    | @redex "⟨" Num "*" Num "⟩"
    | @redex "⟨" "(" Num ")" "⟩";
```

Figure 4 Parenthesization.

The reason is that the associativity (e.g., `left`) and priority annotations used to disambiguate the base grammar, do not transfer to the context grammars. The term `1 - 2 - 3` has two derivations, one through `E "-" Expr`, where `E` derives `Num "-" Num`, and one through `Num "-" E`, where `E` again derives `Num "-" Num`. Although the original term had not been ambiguous, it *became* ambiguous when it was unparsed to text, and reparsed over the context-grammar.

A solution lies in how humans disambiguate expressions: by adding parentheses. This time, however, the parentheses are needed everywhere, since we do not know about the meaning of the terms. The process is illustrated in Figure 5. The starting point is a base grammar L.g, which is used to parse an L-program Foo.L, resulting in a parse tree (indicated by triangles). The context grammar, L+Ctx.g extends the base grammar, and is automatically[3] transformed into a *parenthesized* grammar [19], ⟨L+Ctx⟩.g. The parse tree for Foo.L is unparsed with parentheses, *then* reparsed with ⟨L+Ctx⟩.g, leading to a parse forest Foo.⟨L+Ctx⟩; after removal of the parentheses we obtain a parse forest Foo.L+Ctx, as if it had been parsed over L+Ctx.g, but without the spurious ambiguities. The parenthesizing of the example expression context-grammar is shown in Figure 4, using the special parentheses ⟨ and ⟩.

When split decomposes such a parenthesized term using the parenthesized context grammar as the recipe, spurious ambiguities resulting from missing associativity and priority annotations (so-called *horizontal ambiguities* [4]) are avoided. This is illustrated in Table 1, where the result of splitting before and after parenthesization is shown. Splitting the first term, `(1 - 2) * (2 - 3)`, still produces the (desired) two decompositions. But the second term, `1 - 2 - 3`, now produces the one and only correct decomposition after parenthesizing. The specifications are not polluted by parenthesization, however, as the example specifications below will demonstrate.

[3] This is again an application of Rascal's type reflection capabilities.

Table 1 Difference between splitting before and after parenthesizing.

Input	Before	Parenthesized
`(1 - 2) * (2 - 3)`	`(1 - 2) * (□)` [2 - 3]	⟨⟨(⟨□⟩)⟩ * ⟨(⟨2 - 3⟩)⟩⟩ [1 - 2]
	`(□) * (2 - 3)` [1 - 2]	⟨⟨(⟨1 - 2⟩)⟩ * ⟨(⟨□⟩)⟩⟩ [2 - 3]
`1 - 2 - 3`	□ - 3 [1 - 2]	⟨⟨□⟩ - 3⟩ [1 - 2]
	1 - □ [2 - 3]	

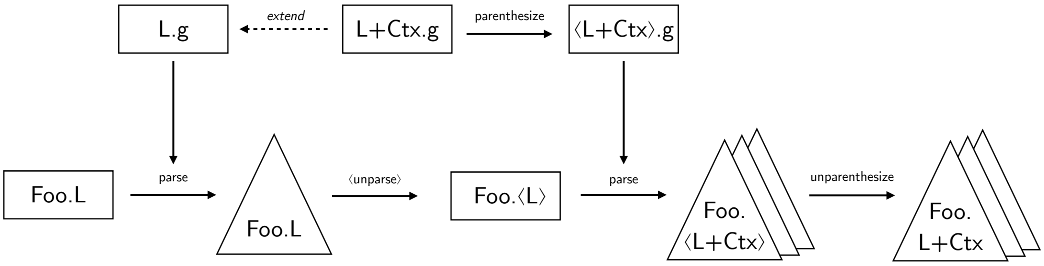


Figure 5 Avoiding spurious ambiguities by parenthesizing context grammars.

4 Example Credex Definitions

4.1 Lambda Calculus

Figure 6 shows the syntax (left) and semantics (right) of the call-by-value lambda calculus. The syntax makes a distinction between expressions (variables, values, and applications), and values (functions, numbers, and the built-in function `+`). The context grammar `E` declares a single context, indicating left-to-right evaluation of a sequence of expressions. Note how `E` "moves" through the sequence by having a prefix of `Value*` and a suffix of `Expr*`. There is only a single redex production: an application of a value to zero or more argument values.

The semantics of lambda calculus is defined using two reduction rules, one for addition, and one for function application. The first rule (`+`) simply performs the addition of two numbers and produces an equivalent expression as a result. The rule for β-reduction substitutes the argument for the parameter of the lambda abstraction using the `subst` function. In turn, this `subst` function (not shown) reuses the generic capture-avoiding substitution facilities of Credex (inspired by [6]).

Consider the term `((λ (x) (+ x 2)) ((λ (x) (+ x 2)) 1))`. Credex comes with a helper function to display the execution trace. The result of the example term is as follows:

```
((λ (x) (+ x 2)) ((λ (x) (+ x 2)) 1))
 └─ ((λ (x) (+ x 2)) ((λ (x) (+ x 2)) 1)) —β→ ((λ (x) (+ x 2)) (+ 1 2))
    └─ ((λ (x) (+ x 2)) (+ 1 2)) —+→ ((λ (x) (+ x 2)) 3)
       └─ ((λ (x) (+ x 2)) 3) —β→ (+ 3 2)
          └─ (+ 3 2) —+→ 5
```

Note that the rendered terms are rendered in the object syntax of the lambda calculus itself[4].

Here's term to illustrate capture-avoiding substitution: `((λ (x) (λ (y) x)) (λ (z) y))`. The variable `y` is free in the top-level argument, so a naïve syntactic substitution would cause capturing. But the result is as follows:

[4] The sequential trace is rendered in an indented fashion to allow for branching in the trace; see Section 4.2.

```
syntax Expr
  = var: Id
  | val: Value
  | app: "(" Expr+ ")";

syntax Value
  = lam: "(" "λ" "(" Id ")" Expr ")"
  | \num: Num
  | add: "+";
```

```
syntax E
  = "(" Value* E Expr* ")"
  | @redex "(" Value Value* ")";

CR red("+", E e, (E)‘(+ <Num n1> <Num n2>)‘)
  = {<e, [Expr]"<toInt(n1) + toInt(n2)>">};

CR red("β", E e, (E)‘((λ (<Id x>) <Expr b>) <Value v>)‘)
  = {<e, subst((Expr)‘<Id x>‘, (Expr)‘<Value v>‘, b)>};
```

Figure 6 Syntax (left) and semantics (right) of the call-by-value lambda calculus.

```
((λ (x) (λ (y) (+ y x))) (λ (z) y))
 └─ ((λ (x) (λ (y) (+ y x))) (λ (z) y)) —β→ (λ (y_) (+ y_ (λ (z) y)))
```

Note how both occurrences of y are renamed to avoid capture.

The substitution facility offered by Credex is parameterized by name analysis of the object language using an embedded Rascal DSL to modularly and concisely express binding relations between declarations and variables. The resulting reference graph is used to detect capturing and rename variables accordingly, similar to the technique of Erdweg et al. [6].

Since Rascal's module system allows extension of both context-free grammars and functions specified in the pattern-based dispatch style (like the function red), semantic specifications using Credex can be modularly extended as well. As an example, consider the extension of the lambda calculus with call/cc . This would consist of a module extending the base-level semantics containing the following code:

```
syntax Value = "call/cc";

CR red("callcc", E e, (E)‘(call/cc <Value v>)‘)
  = {<e, (Expr)‘(<Value v> (λ (<Id x>) <Expr cc>))‘>}
 when
   Id x := fresh((Id)‘x‘, e),
   Expr cc := plug(#Expr, e, (Expr)‘<Id x>‘);
```

The first line extends the syntax of values with the primitive call/cc (it is not a redefinition of Value). The reduction rule labeled callcc extends the red function with the semantics of call-with-current-continuation. Additionally, this shows how contexts can be embedded into terms to model continuations. The result of calling call/cc with a function as argument is to call that function with a lambda modeling the current continuation, cc, which is constructed by plugging a fresh variable x into the E context e. The helper function fresh creates an unique new variable x relative to the context itself. Here's a trace showing call/cc in action:

```
(+ 1 (call/cc (λ (k) (k 2))))
 └─ (+ 1 (call/cc (λ (k) (k 2)))) —callcc→ (+ 1 ((λ (k) (k 2)) (λ (x) (+ 1 x))))
     └─ (+ 1 ((λ (k) (k 2)) (λ (x) (+ 1 x)))) —β→ (+ 1 ((λ (x) (+ 1 x)) 2))
         └─ (+ 1 ((λ (x) (+ 1 x)) 2)) —β→ (+ 1 (+ 1 2))
             └─ (+ 1 (+ 1 2)) —+→ (+ 1 3)
                 └─ (+ 1 3) —+→ 4
```

4.2 Imp: a Simple Imperative Language

Imp is a simple imperative language consisting of arithmetic expressions, boolean expressions, and statements (if-then-else, while, assignment, skip). Its semantics requires a mutable store, which is modeled syntactically as shown in Figure 7. State defines the store as a sequence of zero or more VarInt pairs (mapping identifiers to integers), separated by commas. A configuration Conf is defined as statement Stmt under a certain State. Configurations are the

```
syntax State = "[" {VarInt ","}* "]";
syntax VarInt = Id "↦" Int;
syntax Conf = State "⊢" Stmt;

syntax C = State "⊢" S;
```

```
syntax S
= Id ":=" A
| seq: S ";" Stmt
| "if" B "then" Stmt "else" Stmt "fi"
| @hole "skip" ";" Stmt
| @hole "if" Bool "then" Stmt "else" Stmt "fi"
| @hole "while" BExp "do" Stmt "od"
| @hole Id ":=" Int;
```

Figure 7 Stores and configurations (left) and statement contexts (right).

```
CR red("seq", C c, (S)‘skip; <Stmt s2>‘) = {<c, s2>};
CR red("if-true", C c, (S)‘if true then <Stmt s1> else <Stmt s2> fi‘) = {<c, s1>};
CR red("if-false", C c, (S)‘if false then <Stmt s1> else <Stmt s2> fi‘) = {<c, s2>};
CR red("while", C c, (S)‘while <BExp b> do <Stmt s> od‘)
  = {<c, (Stmt)‘if <BExp b> then <Stmt s>; while <BExp b> do <Stmt s> od else skip fi‘>};
CR red("assign", C c, (S)‘<Id x> := <Int i>‘) = {<c[state=s2], (Stmt)‘skip‘>}
  when isDefined(x, c.state), State s2 := update(x, i, c.state);
```

Figure 8 Statement reduction rules for the simple, imperative Imp language.

top-level terms that will be rewritten. The rule for context `C` simply declares that traversal should always go into the statement part, modeled by context `S`, which further defines the evaluation order of statements, where `A` and `B` represent contexts for arithmetic expressions and boolean expressions respectively.

The reduction rules for the imperative fragment of Imp are shown in Figure 8. The rule for sequencing states that the `skip` statement can be skipped, and execution proceeds with the right-hand side `s2`. The rules for if-then-else reduce to their respective branches based on value of their conditions. The semantics of while-loops is expressed in terms of if-then-else and another while-loop. Finally, assignment to a variable `x` updates the context *itself*, to record the updated value of `x` in the current state, and then reduces to skip. Here's the reduction trace of `x := 1; y := x + 2; if x <= y then x := x + y else y := 0 fi`:

```
[x↦0,y ↦0] ⊢ x:=1; y:=x+2; if x<=y then x:=x+y else y:=0 fi
 └ [x↦0,y ↦0] ⊢ x:=1; y:=x+2; if x<=y then x:=x+y else y:=0 fi —assign→ [x ↦ 1, y ↦0] ⊢ skip; y:=x+2; if x<=y then x:=x+y else y:=0 fi
   └ [x ↦ 1, y ↦0] ⊢ skip; y:=x+2; if x<=y then x:=x+y else y:=0 fi —seq→ [x ↦ 1, y ↦0] ⊢ y:=x+2; if x<=y then x:=x+y else y:=0 fi
     └ [x ↦ 1, y ↦0] ⊢ y:=x+2; if x<=y then x:=x+y else y:=0 fi —lookup→ [x ↦ 1, y ↦0] ⊢ y:=1+2; if x<=y then x:=x+y else y:=0 fi
       └ [x ↦ 1, y ↦0] ⊢ y:=1+2; if x<=y then x:=x+y else y:=0 fi —add→ [x ↦ 1, y ↦0] ⊢ y:=3; if x<=y then x:=x+y else y:=0 fi
         └ [x ↦ 1, y ↦0] ⊢ y:=3; if x<=y then x:=x+y else y:=0 fi —assign→ [x ↦ 1, y ↦ 3] ⊢ skip; if x<=y then x:=x+y else y:=0 fi
           └ [x ↦ 1, y ↦ 3] ⊢ skip; if x<=y then x:=x+y else y:=0 fi —seq→ [x ↦ 1, y ↦ 3] ⊢ if x<=y then x:=x+y else y:=0 fi
             └ [x ↦ 1, y ↦ 3] ⊢ if x<=y then x:=x+y else y:=0 fi —lookup→ [x ↦ 1, y ↦ 3] ⊢ if 1<=y then x:=x+y else y:=0 fi
               └ [x ↦ 1, y ↦ 3] ⊢ if 1<=y then x:=x+y else y:=0 fi —lookup→ [x ↦ 1, y ↦ 3] ⊢ if 1<=3 then x:=x+y else y:=0 fi
                 └ [x ↦ 1, y ↦ 3] ⊢ if 1<=3 then x:=x+y else y:=0 fi —leq→ [x ↦ 1, y ↦ 3] ⊢ if true then x:=x+y else y:=0 fi
                   └ [x ↦ 1, y ↦ 3] ⊢ if true then x:=x+y else y:=0 fi —if-true→ [x ↦ 1, y ↦ 3] ⊢ x:=x+y
                     ├ [x ↦ 1, y ↦ 3] ⊢ x:=x+y —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=1+y
                     │ └ [x ↦ 1, y ↦ 3] ⊢ x:=1+y —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=1+3
                     │   └ [x ↦ 1, y ↦ 3] ⊢ x:=1+3 —add→ [x ↦ 1, y ↦ 3] ⊢ x:=4
                     │     └ [x ↦ 1, y ↦ 3] ⊢ x:=4 —assign→ [x ↦ 4, y ↦ 3] ⊢ skip
                     └ [x ↦ 1, y ↦ 3] ⊢ x:=x+y —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=x+3
                       └ [x ↦ 1, y ↦ 3] ⊢ x:=x+3 —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=1+3
                         └ [x ↦ 1, y ↦ 3] ⊢ x:=1+3 —add→ [x ↦ 1, y ↦ 3] ⊢ x:=4
                           └ [x ↦ 1, y ↦ 3] ⊢ x:=4 —assign→ [x ↦ 4, y ↦ 3] ⊢ skip
```

Note how the concrete syntax for the store naturally transfers to rendering of terms. Note also the split in the execution trace, since Imp does not enforce an evaluation order for addition. Both branches evaluate to the same configuration, as expected.

4.3 QL: a Language for Questionnaires

The Questionnaire Language (QL) [7,8] is a DSL with a non-standard execution model. It interesting for three reasons: first, it models an event-based system, a questionnaire form where users interactively change inputs of values. Second, QL allows declare-after-use of questions: a computed question may refer to the value of another question occurring *later*

```
// expressions are immediately evaluated to values
CR red("eval", C c, (E)`<Expr e>`) = {<c, (Expr)`<Value val>`>}
when
  value v := eval(e, c.ui), Value val := [Value]"<v>";

// a user action updates the ui state and then expands to a block of statements
// to reconcile the UI with the consequences of the update
CR red("update", C c, (S)`update(<Id x>, <Value v>)`)
  = {<c[ui=updateVal(c.ui, x, v)], makeBlock(c.qs, c.ui, x)>};

// dealing with unit of statement sequencing
CR red("done", C c, (S)`{ { } <Stmt* s2>}`) = {<c, (Stmt)`{ <Stmt* s2>}`>};

// updating a question to a value that is the same as the old value is a no-op
CR red("val-same", C c, (S)`val(<Id x>, <Value v>, <Value old>)`) = {<c, (Stmt)`{}`>}
  when old == v;

// otherwise, updating is equivalent to a user action
CR red("val-diff", C c, (S)`val(<Id x>, <Value v>, <Value old>)`)
  = {<c, (Stmt)`update(<Id x>, <Value v>)`>} when old != v;

// updating visibility modifies the UI
CR red("vis", C c, (S)`vis(<Id x>, <Bool b>)`) = {<c[ui=updateVis(c.ui, x, b)], (Stmt)`{}`>};
```

Figure 9 Reduction rules for QL.

in the form. This requires a fixpoint computation in the style of spreadsheets. Finally, the consequences of user actions are not limited to the state, but also affect the UI; in other words, the semantic domain is complex.

The core reduction rules for QL are shown in Figure 9. First, QL features expressions for definining computed questions and conditional visibility of questions. Since such expressions are semantically rather uninteresting, they are evaluated in one step, using an existing interpreter (rule `eval`). This shows how CREDEX integrates well with other language engineering components within Rascal.

The semantics starts off with a user action `update(`x`, `v`)` modifying the value of a (non-computed) question. The reduction rule (`update`) for this redex expands to a sequence of derived statements (using `makeBlock`; not shown), representing the "update plan" according to the questionaire. Modifications to the state check whether a value did change, and if so, trigger new `update`-statements (rules lstlineval-same and `val-diff`). As result, execution continues till the state reaches a fixed point.

5 Instead of conclusion

In this short paper we have presented CREDEX, a library in Rascal for defining small-step semantics. Next to the basics detailed above, CREDEX comes with browser-based tools for visualizing execution graphs, interactive step-wise debugging, and functionality for randomized sentence generation. Further research directions include: a precise comparison to the matching algorithm of Redex [14], and investigating how CREDEX can be combined with Rascal's typechecking library, TYPEPAL.

There are many tools out there to define and execute formal semantics (e.g., [1–3, 9, 20, 23–26], and others). CREDEX is unique in that it takes the basic execution model of Redex (but using concrete syntax), it is modular from the start, it employs parsing for context-redex decomposition, and it integrates well with the Rascal language workbench. As such, it brings semantics engineering a small step closer to language engineering.

References

1 Yves Bertot. A short presentation of coq. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 12–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

2 L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. Executable component-based semantics. *Journal of Logical and Algebraic Methods in Programming*, 103:184–212, 2019. doi:10.1016/j.jlamp.2018.12.004.

3 L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 8–11, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892664.2893464.

4 Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, 2010. doi:10.1016/j.scico.2009.11.002.

5 Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *PLDI'19*, pages 1133–1148. ACM, June 2019. doi:10.1145/3314221.3314601.

6 Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-avoiding and hygienic program transformations. In Richard E. Jones, editor, *ECOOP'14*, volume 8586 of *Lecture Notes in Computer Science*, pages 489–514. Springer, 2014. doi:10.1007/978-3-662-44202-9_20.

7 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 197–217, Cham, 2013. Springer International Publishing.

8 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014). doi:10.1016/j.cl.2015.08.007.

9 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. MIT Press, 2009.

10 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.

11 Jan Heering, Paul Klint, et al. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, 2000.

12 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, October 2010. doi:10.1145/1932682.1869497.

13 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *POPL'12*, pages 285–296. ACM, 2012. doi:10.1145/2103656.2103691.

14 Casey Klein, Jay A. McCarthy, Steven Jaconette, and Robert Bruce Findler. A semantics for context-sensitive reduction semantics. In *APLAS'11*, volume 7078 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2011. `doi:10.1007/978-3-642-25318-8_27`.

15 Paul Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993. `doi:10.1145/151257.151260`.

16 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009. `doi:10.1109/SCAM.2009.28`.

17 Jacob Matthews and Robert Bruce Findler. An operational semantics for Scheme. *Journal of Functional Programming*, 18(01):47–86, 2008.

18 Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. *A Visual Environment for Developing Context-Sensitive Term Rewriting Systems*, pages 301–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. `doi:10.1007/978-3-540-25979-4_21`.

19 Robert McNaughton. Parenthesis grammars. *J. ACM*, 14(3):490–500, July 1967. `doi:10.1145/321406.321411`.

20 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2628136.2628143`.

21 Gordon D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004. Structural Operational Semantics. `doi:10.1016/j.jlap.2004.03.009`.

22 Gordon D. Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Structural Operational Semantics. `doi:10.1016/j.jlap.2004.05.001`.

23 Grigore Rosu. K – a semantic framework for programming languages and formal analysis tools. In Doron Peled and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, NATO Science for Peace and Security. IOS Press, 2017.

24 Carsten Schürmann. The twelf proof assistant. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 79–83, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

25 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9):1–12, October 2007. `doi:10.1145/1291220.1291155`.

26 Vlad A. Vergu, Pierre Neron, and Eelco Visser. Dynsem: A DSL for dynamic semantics specification. In *RTA'15*, pages 365–378, 2015. `doi:10.4230/LIPIcs.RTA.2015.365`.

27 Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher Order Symbol. Comput.*, 14(4):387–409, December 2001. `doi:10.1023/A:1014408032446`.

Context in Parsing: Techniques and Applications

Eric Van Wyk
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

Abstract

This paper discusses two approaches to parsing: Eelco Visser's scannerless generalized LR parsing and our context-aware scanning paired with deterministic LR parsing. We compare the underlying techniques, specifically how parser context is used to disambiguate lexical syntax, and their use in the context of language evolution and composition applications. We also reflect on the many discussions shared with Eelco on these topics, and on our shared realization that our different assumptions about the contexts in which our approaches were used drove and justified the technical decisions made in each.

2012 ACM Subject Classification Software and its engineering → Syntax; Software and its engineering → Parsers; Software and its engineering → Domain specific languages

Keywords and phrases Parsing, Generalized LR Parsing, Context-aware Scanning

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.30

1 Introduction

One thing we have all missed due to the Covid pandemic, and fear missing in the future due to reduced travel for climate change reasons, is the *hallway track* at conferences. These opportunities to talk with fellow computer scientists are an important part of research and helpful in driving it forward. They are also a lot of fun. These informal interactions are, at least so far, difficult to recreate online at virtual events. Other losses include the highly-interactive discussions and presentations at workshops and smaller conferences like the discontinued, but much loved, Workshop on Language Descriptions, Tools, and Applications (LDTA), its vibrant replacement Software Language Engineering (SLE), and the occasional satellite workshops such as Parsing@SLE and OOPSLE.

The discussions at these events – in meeting rooms, in the hallway, or at dinner in the evening – are almost always lively ones. Over the years, the conversations I had with Eelco Visser stand out as some of the most enjoyable and thought provoking. Eelco and I were friendly intellectual sparring partners, each of us advocating for our own approach to scanning and parsing. Eelco had developed scannerless generalized-LR parsing [25, 26] (SGLR) while August Schwerdfeger and I had developed context-aware scanning used with deterministic LR parsers [24, 18] (CAS+LR).

Both of our approaches treat the processes of recognizing lexical syntax (scanning) in the text and determining its phrase structure (parsing) as interdependent ones. This was in contrast to the traditional approach that sees these two tasks as separate; in this view a scanner can first consume the input text and generate a complete sequence of lexical tokens. These tokens are then provided to the parser. This separation of lexical syntax and phrase (context-free) syntax is a natural one. In a program text file the sequence of characters is naturally seen as a sequence of words (identifiers, keywords, punctuation, numeric literals, etc.) interspersed with whitespace and comments. It is the scanner's job to recognize these words in the sequence of characters, discarding whitespace (when it is not relevant) and comments. These lexical symbols are commonly specified using regular expressions. Note that we will use the terms *scanner* and *scanning* as shorthand for the component for, and process of, recognizing lexical syntax even in SGLR despite there being no separate scanner. The parser consumes these lexical tokens and attempts to recognize the underlying

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 30; pp. 30:1–30:10
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

phrase structure, for example, that a *while-loop* begins with a `while` keyword, followed by an expression acting as the loop condition, and then a statement wrapped in curly braces. Context-free grammars are typically used to define the phrase structure of the language.

Context in lexical syntax – a shared divergence

In both SGLR and CAS+LR, the state of the parser provides contextual information to the process for recognizing lexical syntax. Both use LR parsing [14, 1] and thus the current state of the LR parser automatically provides this notion of context. This is so that in different parsing contexts the same sequence of characters can be recognized in different ways, as different lexical tokens.

In his paper "Pure and Declarative Syntax Definition: Paradise Lost and Regained" from Onward! 2010 [10], Eelco and his coauthors explain this need for context with a few examples. One is the challenge of recognizing the lexical syntax in the text "`array [1..10] of integer`". A fragment of a specification using a context-free grammar and regular expressions to define this can be seen in Figure 1. Nonterminals `Type` and `Expr` derive type expressions and value expressions as expected. Lexical syntax would be defined as expected as well. Keywords and punctuation (written between single quotes) are recognized as such; the only non-constant lexical syntax (of interest here) is for integer and floating point literals. The first being a non-empty sequence of digits, the second being a sequence of digits with a trailing, leading, or contained period ("`.`").

In the string "`array [ 1..10 ]`" we need to recognize "`1`" as an integer literal and not recognize "`1.`" as a floating point literal. A traditional scanner would prefer the longer, yet incorrect, match. If the parser context can be used, then the fact that floating-point literals are not allowed in between the square brackets of an array type declaration can disallow the longer match of "`1.`" and instead correctly recognize the shorter integer literal of "`1`".

```
Type ::= 'integer'
      |  'float'
      |  'array' '[' IntLiteral '..' IntLiteral ']' 'of' Type

Expr ::= IntLiteral
       | FloatLiteral
       | ...

IntLiteral /[0-9]+/
FloatLiteral /([0-9]+\.)|(\.[0-9]+)|([0-9]+\.[0-9]+)/
```

Figure 1 A partial specification for recognizing `array [1..10] of integer`.

Eelco's paper on parsing AspectJ [2], a language that introduced the notion of aspects and aspect-oriented programming [12, 11] to Java, provides the first declarative specification of the language's concrete syntax. As an example, consider the admittedly contrived and abbreviated AspectJ fragment shown in Figure 2. It contains a standard Java class `Sample` with 4 integer fields, "getter" methods for two of them, and a method named `after`. The aspect `Example` will increment the `count` field on an object `o` of type `Sample` *after* a call to either of the getter methods. The pattern "`get*`" on line 9 matches any calls to methods beginning with the letters "`get`" and the increment action takes place after the call to any matched methods.

```
1 class Sample {
2   int get, count, size, shape;
3   int getSize () { ... }
4   int getShape () { ... }
5   boolean after() { return (get*3 > 15) ; }
6 }
7 aspect Example {
8   ...
9   after(): o.get*()  { o.count ++; }
```

Figure 2 An abbreviated example of AspectJ.

We are not so interested in the semantics of AspectJ and more concerned with how its lexical and context-free syntax can be recognized. The first point to note is that "after" is used as an identifier on line 5 and as a keyword on line 9. AspectJ introduces new keywords inside an "aspect" block, but these are still legal as identifiers elsewhere. On line 5 we see the text "get*" which should be recognized as two lexical symbols: the identifier "get" and a multiplication sign "*". But in the context of the AspectJ pattern on line 9 this same text should instead be seen as pattern matching any identifier beginning with "get". Again, we see that in different parts of the program – that is, in different contexts – the same text is recognized as different lexical symbols.

Both SGLR and CAS+LR use parser context to drive the recognition of lexical syntax and thus both directly and easily handle the examples above. The CAS+LR specification of AspectJ [16, 17] was developed in response to Eelco's paper on this topic as we wondered if a deterministic specification was possible in our approach. The techniques these two approaches use are similar in some regards, but different in others. SGLR uses generalized LR parsing and can thus parse the entire class of context-free grammars, while our implementation of CAS+LR uses an LALR(1) parser. In SGLR there is no scanner. Parsing is done down to the character level, thus the term "scanner" is a misnomer since it is all just parsing. Thus parser context is directly used in recognizing lexical syntax. The scanner in CAS+LR is called by the parser each time a new token is needed and it tells the scanner which lexical symbols are currently valid. The scanner uses this to return only lexical symbols that are valid for the current parsing context.

Context in applications

Despite the similarities in recognizing lexical syntax, Eelco and I frequently debated the relative merits of our respective approaches. The answer to the (usually) unspoken question of which one is "better" also relies on a notion of context. The short answer is that neither is generally better; there are contexts in which each might be the wiser choice and these different contexts justify the technical decisions made in each approach.

The different assumptions that underpin our different approaches can be most directly seen in a sentence from the Eelco's "Paradise Regained" paper [10] mentioned above. This paper embraces the freedom of expression afforded papers at the Onward! conference and describes the use of parsing that requires a deterministic subclass of context-free grammars such as LALR(1) as a Biblical "fall from grace." The paper is a joy to read. Eelco lays out the case that language engineers (those who write grammars and build parsers) should be *oblivious* to the parsing algorithms used in parser generators and be *free* to use any context-free grammar. This is his "Garden of Eden." The serpent that leads to the fall from grace and exiting of the garden is the pursuit of speed and efficiency that forces one into limited deterministic subclasses such as LALR(1).

The linchpin in understanding the different paths that SGLR and CAS+LR have taken can be seen in Eelco's proclamation in his paper [10, page 918] that in the Garden of Eden

> "The language engineers were software engineers and
> the software engineers were language engineers."

In this paper we consider the ramifications of this proclamation for SGLR and how CAS+LR takes a different path by not adhering to it. In Section 2 we revisit some of the technical aspects of SGLR and CAS+LR and discuss the similarities and differences between the two approaches. The focus here is on their use of parsing context in recognizing lexical syntax. In Section 3 we explore the different scenarios, that is application contexts, in which the two approaches best serve their users and discuss how these contexts and assumptions inherent in them led to the different techniques used in each. Section 4 concludes.

2 Context in scanning and parsing techniques

In this section we briefly describe the fundamental techniques in SGLR and CAS+LR, discussing a few relative merits. The description is brief; the cited papers provide the details.

Scannerless Generalized LR Parsing: SGLR

The main points of interest in scannerless generalized LR parsing (SGLR) are right there in the name: the use of generalized parsing techniques and parsing down to the character level so that a separate scanner is not needed.

Generalized parser generators can build parsers for any context-free grammar, thus freeing their users from the restrictions of writing only grammars within a deterministic subclass such as LR(k) or LALR(k). Since all grammars are allowed, even ambiguous ones, users must be alert to this possibility. Ambiguity can often be resolved by refactoring the grammar but this can lead to unnatural and convoluted grammars. For example, ambiguities in expressions with infix operators can be handled by breaking a single *expression* nonterminal into layers (with names such as *term* and *factor*) corresponding to operator precedence. The famous dangling-else ambiguity splits a *statement* nonterminal into so-called *open* and *closed* varieties that is even more convoluted than the expression refactoring.

The "pure" solution to this problem, as advocated by Eelco [10], is to use disambiguation filters. As the name suggests, when an ambiguous parse produces two (or more trees) for the input text, the filters remove the undesired trees, keeping the desired one. These filters can take the form of stating a preference for one grammar production over another, or higher level filters for specifying infix operator precedence and associativity. In SDF, the grammar formalism originally used in SGLR, one can write the specification on the left of Figure 3 (as shown in [10, page 925]) instead of the more verbose one on the right.

A sequence of Eelco's papers provides a compelling case for this approach [13, 21, 4, 5]. It allows one to write more natural grammars, free of the convolutions required to fit the concrete grammar into a deterministic subclass. The resulting concrete syntax trees generated by the parser are simple and thus there is often no need for a second simplified "abstract" tree representation for semantic analysis.

The "scannerless" in SGLR means what it says; there is no scanner. Instead it is parsing all the way down to the character level. As Eelco says in the "Paradise Regained" paper, "and the words were trees." This results in a uniform specification for recognizing both lexical

```
E "*" E -> E {left} >
E "+" E -> E {left}
Literal -> E
```

```
E "+" T -> E
T -> E
T "*" F -> T
F -> T
Literal -> F
```

Figure 3 Grammar disambiguation by filters (left) and nonterminal refactoring (right).

and context-free syntax. Whitespace symbols are ignored, and parsing of comments allows for nested comments – something that is not possible when using only regular expressions to specify lexical syntax. Here disambiguation filters are used to ensure that the traditional longest match ("maximal munch") behavior of matching lexical syntax is enforced.

Returning to the examples from Section 1 we can now see how SGLR uses the context of the parser to properly recognize lexical syntax. In "`array [ 1..10 ]`" the bottom-up behavior of SGLR may recognize "`1`" as an integer literal and also recognize "`1.`" as a floating point literal, but this temporary ambiguity is resolved in the context of "`array [`" since only the former can be used to form a syntactically valid tree. The latter leads to a syntax error and thus that parsing thread is abandoned. Here the parser context recognizing lexical syntax is implicit in that it is all part of a single parser. In the AspectJ example [2] the same techniques are applied. For example, the text "`after`" on line 9 may be recognized as both a Java identifier and an AspectJ keyword, but only the keyword is valid at the beginning of this line, and thus it is used and the identifier tree is discarded.

Context-aware Scanning and LR Parsing: CAS+LR

The primary insight of context-aware scanning [17, 24] is (again) that the parser context can be used by the scanner to be more discriminating in how it recognizes lexical syntax. Both a scanner and parser are used in this approach, but there is a modification in how these two components interact. Lexical syntax is still defined by regular expressions, and a scanner is generated by converting these to a deterministic finite automata (DFA). The difference is that a context-aware scanner has two types of input when called by the parser to produce the next token. The first is the text being scanned. The second is the set of terminal symbols that are *valid* in the current parser context. When context-aware scanning is used with an LR parser, this set of valid terminal symbols is the set of terminals in the current LR parse table state that have entries of *shift*, *reduce*, or *accept*, but not *error*. While the details of how the context-aware scanner is constructed and makes use of this input is to be found in the papers cited above, the idea is a simple one. Where a traditional scanner follows the longest-match rule to prefer a longer token over a shorter one, a context-aware scanner will prefer shorter valid tokens over longer invalid ones.

In the "`array [1..10] of integer`" example, after the parser has shifted the "`array`" and the "`[`" tokens it is a state in which `IntLiteral` has an action of *shift* but `FloatLiteral` has an *error* action. Thus the context-aware scanner only returns the former, and does not consume the "`.`" character and thus does not recognize the latter. The same process plays out in the AspectJ example. In Figure 2, the "`after`" on line 9 is in a parser context in which the AspectJ keyword `after` is valid but the Java identifier is not. The scanner DFA reaches a final state labeled by both terminal symbols and returns the one that is in the valid terminal set for that call to the scanner.

3 Application Contexts: Language Evolution and Composition

The various merits, advantages, and disadvantages of SGLR and CAS+LR were the fodder for many discussions that Eelco and I enjoyed. While we occasionally talked about performance, our main topic was the issue of ambiguity in grammars. It was at one Parsing@SLE, in Indianapolis I believe, that I asked Eelco if the benefit of using an LALR(1) grammar and thus knowing, statically, that the grammar was unambiguous was like static typing in programming languages. Is it not better to know that an ambiguous parse is impossible in the same way that static type systems ensure that run-time type errors are impossible? Eelco agreed that it was better to know desirable properties statically, but that this was more like statically checking for program termination than statically checking for type errors. Many programmers use statically typed languages because the effort to show a program is well-typed is not an onerous one. However, showing that a program terminates is onerous and there is little tool support for doing so. I agreed, but felt that ambiguities can be more difficult to identify than non-termination. He felt the cost of contorting a grammar into a deterministic class like LALR(1) was too high a price to pay for that guarantee. It seemed that perhaps it was simply a matter of personal preference – balancing static guarantees and performance against flexibility in expression and ease of use.

We were both interested in domain-specific languages (DSLs), and extensible languages in which programmers add domain-specific language features to a host language such as Java or C. In this context some clarity about the technical decisions made by the two approaches can be found. Consider a language component introducing syntax for SQL database queries to Java. It may allow for syntax like the following for establishing a connection named `mydb` to a database server and ensuring that a `person` table there has the expected columns:

```
connection mydb "jdbc:derby:./person_db"
   with table person [ person_id INTEGER, first_name VARCHAR ];
```

Another language extension may introduce condition tables, a construct useful for understanding complex boolean expressions. A simple example can be seen below:

```
boolean b = table ( c1 :  T F
                    c2 :  F * );
```

This sets a boolean identifier `b` to the value of the condition table. The table will evaluate to true if condition `c1` is true (`T`) and `c2` is false (`F`) or if `c1` is false and `c2` has either value (`*`). This condition is equivalent to `(c1 && !c2) || (!c1)` and may be translated down to this expression or an equivalent one that avoids evaluating `c1` twice. Tables like these are used in modeling language such as RSML^{-e} [19] and SCR* [6] where complex Boolean conditions need to be understood by software stakeholders. This is another example in which parser context disambiguates lexical syntax, in this case to recognize "`table`" as a keyword token from the correct extension specification. Eelco's MetaBorg approach using SGLR allows language users to extend Java with new syntax similar to the examples above [3] while we did it using CAS+LR in for Java in our ableJ [23] system and later for C in ableC [8].

Eelco's research, embodied in the Spoofax language workbench [9] and its collection of language processing tools, allows software engineers (the language users) to have a hand in prototyping, designing, and implementing the languages they wanted to use to solve their problems. This can be done using extensible languages, as described above, in developing new domain-specific language features and composing them with others to create an extended language tailored for a particular task. Eelco's work also enables and encourages engineers to design their own stand-alone DSLs. In creating a new language it is not uncommon for the (composed) context-free grammar to be unambiguous even if it is not in a deterministic subclass like LALR(1). If it is ambiguous, the ambiguities are seen as bugs. One can still use the generated parser and simply treat an ambiguous parse as a syntax error. If this kind of error occurs too frequently, then one may work to resolve the ambiguity in the specification by adding or modifying a disambiguation filter, such as those shown on the left of Figure 3. This approach allows an engineer to easily prototype a new language, a central focus of Eelco's work. The effort, sometimes a considerable one, to transform an initial grammar to be a member of a deterministic subclass can be avoided. This view rests on his proclamation that the roles of language engineer and software engineer should be interchangeable, and is realized by the technical choices made in SGLR and its supporting tools.

My students and I were primarily interested in language evolution and feature composition in a different context – one in which the roles of language engineer and software engineer are kept as distinct. That is, we start with a set of assumptions different from those encapsulated in the proclamation from Eelco's "Paradise Regained" paper. In this view, language extensions are independently developed by authors (language engineers) familiar with context-free grammars and parser-generator tools (and the attribute grammar formalisms used to specify the semantics of the language features [22]). The users of language extensions however – the software engineers composing a language from some set of extensions – need *not* be familiar with the parsing techniques. This view acknowledges that some software engineers may simply not want to know about grammars and parsing (as bizarre as it may seem to many readers of this paper.) Software engineers are not only oblivious to any underlying parsing algorithms, they are also oblivious to parsing and context-free grammars. Our interest was in seeing how far we could go with this different set of assumptions. This view was realized in the ABLEC extensible specification for C and a collection of modular and reliably composable language extensions [8, 7].

For this approach to be feasible, language extensions must compose with the host language automatically, and the resulting composition must be well-formed. For concrete syntax specifications, this requires the composed context-free grammar to be unambiguous and for there to be no lexical ambiguities in the lexical syntax. Thus we developed a *modular determinism analysis* [18, 17] for context-free and lexical syntax provides a guarantee for composed specifications: if the extensions independently pass the analysis, then the specification automatically composed from the user-selected extensions would be deterministic, specifically, LALR(1) and the lexical specification would have no lexical ambiguities.

Composing context-free grammars and regular expressions is simply a matter of taking the union of the sets of productions, nonterminals, and terminals (with associated regular expressions) in the separate specifications, denoted as $\cup^*$ below. We are concerned with some specification C that is the composition of a host language H and set of independently-developed extensions $\{E_1, ..., E_n\}$, that is $C = H \cup^* E_1 \cup^* ... \cup^* E_n$. The property provided by the modular determinism analysis can be expressed as

$$(\forall i \in [1, n] \ .det_m(H, E_i)) \ \Rightarrow \ det(H \cup^* E_1 \cup^* ... \cup^* E_n)$$

This property states that if each extension E_i satisfies the modular composability criteria (det_m) with respect to the host language H, then the lexical and context-free syntax of the composition of all components ($H \cup^* E_1 \cup^* ... \cup^* E_n$) is also deterministic (det). For context-free syntax, the modular analysis det_m checks certain characteristics of the grammar $H \cup^* E_i$ to ensure that the composition of all the grammars ($H \cup^* E_1 \cup^* ... \cup^* E_n$) will be LALR(1) and thus non-ambiguous. First it requires $H \cup^* E_i$ to be LALR(1). Additionally det_m requires any production in E_i with a host language nonterminal on the left-hand side to have, as the first symbol on its the right-hand side, a new so-called *marking* terminal [18] introduced by that extension. The `table` keyword from the condition table example above is a marking terminal. It also checks that the follow sets of host nonterminals in $H \cup^* E_i$ do not exceed those from H alone. Additional checks [18] on the LR-automata generated from $H \cup^* E_i$ ensure that each extension creates an effectively isolated part of the eventually composed LR-automata that corresponds to having no conflicts in the parse table.

For lexical syntax, determinism requires that in any parsing context only one terminal symbol be matched. The primary requirement is that for any two different terminals t_1 and t_2, if (*i*) they are both valid for some parser state p of the LR-parse table and (*ii*) t_1 and t_2 both label a final state in the scanner DFA (meaning both could be returned by the scanner), then one has lexical precedence over another. An example of this is keyword terminals being preferred over identifier terminals. This ensures that SQL `table` keyword will never conflict with the condition-table `table` keyword from above. The isolation of LR-automata for different extensions, as checked by det_m, ensures that this determinism condition is met in the final composition for all LR parse states except for those states that are considered to be "host" states. For example, if a third extension introduces a marking terminal matching "`table`" then there is a lexical ambiguity between two extensions that cannot be detected by the analysis. Fortunately, these ambiguities can be easily resolved by the software engineer composing the extensions (with a bit of tool support) to require a short disambiguating prefix be written before each use of "`table`". This is similar to the Java requirement that if two imported packages define the same type, then some form of qualified name be used to distinguish them. The full description of the analysis can be found in earlier works [18, 17].

The point of this modular analysis is that the cost of being constrained to deterministic grammars is paid by language engineers. This has not, in Eelco's words, "delivered language engineers out of the slavery to parser generators" as they may still resort to the "process of trial-and-error" to "simply torture the parser definition until it confesses" and satisfies the analysis requirements. A significant benefit, however, is reaped by the software engineers *choosing* a set of extensions. They need not know anything about grammars or parsing but are assured that the syntax specifications of the extensions they have chosen will be automatically composed and guaranteed to form an unambiguous specification. This provides the freedom to freely pick the independently-developed language extensions they desire.

It is clear then that these different contexts – in which software engineers *are* or *are not* language engineers – drove the technical decisions made in both SGLR and CAS+LR.

4 Conclusion

We have compared the SGLR and CAS+LR approaches to parsing: the underlying techniques used by both, and the application contexts in which they each shine. Eelco and I eventually realized that we each had a different view of language evolution and composition and that each had developed tools and techniques that supported our individual views. After this, our conversations shifted to focus on other problems in the field of software language engineering, including some on Eelco's influential scope graphs [15, 20].

That said, I do miss our "pugnacious" debates about parsing. Eelco was a wonderful sparring partner in these: committed, insightful, helpful, and caring. It is a great joy to have someone like him who pushes you to think more carefully and clearly about your work and helps you better understand the context in which it resides. He will be sorely missed.

References

1 A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

2 Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for aspectJ. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 209–228. ACM, 2006. doi:10.1145/1167473.1167491.

3 Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, pages 365–383. ACM, 2004. doi:10.1145/1028976.1029007.

4 Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Deep priority conflicts in the wild: a pilot study. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE*, pages 55–66. ACM, 2017. doi:10.1145/3136014.3136020.

5 Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Towards zero-overhead disambiguation of deep priority conflicts. *Programming Journal*, 2(3):13, 2018. doi:10.22152/programming-journal.org/2018/2/13.

6 C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS)*, 1995.

7 Ted Kaminski. *Reliably Composable Language Extensions*. PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2017. URL: http://hdl.handle.net/11299/188954.

8 Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98:1–98:29, October 2017. doi:10.1145/3138224.

9 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, OOPSLA. ACM, 2010. doi:10.1145/1869459.1869497.

10 Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Onward!'10, pages 918–932. ACM, 2010. doi:10.1145/1869459.1869535.

11 G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 Object–Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001. doi:10.1007/3-540-45337-7_18.

12 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object–Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997. doi:10.1007/BFb0053381.

13 Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

14 D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. doi:10.1016/S0019-9958(65)90426-2.

15 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

16 August Schwerdfeger. A declarative specification of a deterministic parser and scanner for AspectJ. Technical Report 09-007, University of Minnesota, March 2009. URL: https://hdl.handle.net/11299/215794.

17 August Schwerdfeger. *Context-Aware Scanning and Determinism-Preserving Grammar Composition, in Theory and Practice.* PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2010. URL: http://purl.umn.edu/95605.

18 August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210, New York, NY, USA, June 2009. ACM. doi:10.1145/1542476.1542499.

19 Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, September 1999. doi:10.1145/318774.318940.

20 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

21 Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002. doi:10.1007/3-540-45937-5_12.

22 Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010. doi:10.1016/j.scico.2009.07.004.

23 Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599. Springer, 2007. doi:10.1007/978-3-540-73589-2_27.

24 Eric Van Wyk and August Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, pages 63–72, New York, NY, USA, 2007. ACM. doi:10.1145/1289971.1289983.

25 Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, August 1997. URL: https://eelcovisser.org/publications/1997/Visser97-SGLR.pdf.

26 Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997. URL: https://eelcovisser.org/publications/1997/Visser97.pdf.

Comparing Bottom-Up with Top-Down Parsing Architectures for the Syntax Definition Formalism from a Disambiguation Standpoint

Jurgen J. Vinju
NWO-I Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
TU Eindhoven, The Netherlands

Abstract

Context-free general parsing and disambiguation algorithms are threaded throughout the research and engineering career of Eelco Visser. Both our Ph.D. theses featured the study of "disambiguation." Disambiguation is the declarative definition of choices among different parse trees, derived using the same context-free grammar, for the same input sentence.

This essay highlights the differences between syntactic disambiguation for context-free general parsing in a top-down architecture and a bottom-up architecture. The differences between top-down and bottom-up are mainly observed as practical aspects of the software architecture and software implementation. Eventually, the concept of data-dependent context-free grammar brings all engineering perspectives of disambiguation back into a conceptual (declarative) framework independent of the parsing architecture. The novelty in this essay is the juxtaposition of three general parsing architectures from a disambiguation point of view: SGLR, SGLL, and DDGLL. It also motivates design decisions in the parsing architectures for SDF{1,2} and Rascal with previously unpublished detail. The essay falls short of a literature review and a tool evaluation since it does not investigate the disambiguation methods of the many other parser generator tools that exist. The fact that only the implementation algorithms are different between the compared parsing architectures, while the syntax definition formalisms have practically the same formal semantics for historical reasons, nicely "isolates the variable" of interest.

We hope this essay lives up to the enormous enthusiasm, curiosity, and drive for perfection in syntax definition and parsing that Eelco always radiated. We dearly miss him.

2012 ACM Subject Classification Software and its engineering → Syntax

Keywords and phrases parser generation, context-free grammars, GLR, GLL, algorithms, disambiguation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.31

Acknowledgements This essay would not have been possible without the past teamwork with Eelco Visser, Paul Klint, Jan Heering, Jeroen Scheerder, Mark van den Brand, Chris Verhoef, Alex Sellink, Pieter Olivier, Hayco de Jong, Georgios (Rob) Economopoulos, Martin Bravenboer, Tijs van der Storm, Joost Visser, Merijn de Jonge, Jørgen Iversen, Arnold Lankamp, Ali Afroozeh, Anastasia Izmaylova, Bas Basten, Martin Bravenboer, Adrian Johnstone and Elizabeth Scott on the topic of context-free general parsing and disambiguation and supporting technologies. However, any error or inconsistency in the following is all mine.

1 Introduction

This essay focuses on qualitative differences in the design and implementation of Syntax Definition, Parser Generation, and Disambiguation between two classes of Parsing algorithms: GLR and GLL. Disambiguation is a function of an entire Parsing Architecture with the goal of reducing the set of Parse Trees (the Parse Forest) to exactly one using declarative definitions. Extensions of the Syntax Definition Formalisms (languages for context-free grammars) allow expressing preferences between different Parse Trees as produced by the Grammar. It is

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 31; pp. 31:1–31:16
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

helpful to see Disambiguation as orthogonal to Parsing, where the latter produces Parse Trees and the former removes them again. However, in actual Parsing Architectures, this distinction is virtually invisible due to efficiency considerations. In this essay, we emphasize declarative and correct parsing over efficiency.

1.1 History of Disambiguation with the SDF

A brief and selective history of disambiguating context-free grammars written in the "Syntax Definition Formalism" (SDF) is due first. We focus on the parsing architectures of the Syntax Definition Formalism (SDF) [18, 15, 19] and its later incarnations SDF2 [26, 45] (a part of the "new" ASF+SDF Meta-Environment [7, 12] and of StrategoXT [13]), and its further parallel offspring in Rascal [24, 25] and Spoofax [22] (SDF3 [4]). This history motivates the comparison of parsing architectures later and establishes their origins and dependencies for the sake of full disclosure. We are not comparing independently designed artifacts.

Already in the first SDF from the early 1980's its users wrote context-free grammars in a BNF-like format [18], plus regular extensions such as lists and optionals (a.k.a. "EBNF"), plus disambiguation declarations. The non-terminal notation of SDF was taken from the meta notation used in Paul Klint's PhD thesis [23]. These definitions were then used to generate parsers and other useful language tooling such as unparsers, syntax highlighters, structure editors, etc. SDF used general parsing algorithms from the very start. Initially, it was based on Jay Earley's algorithm [16], but SDF switched to Tomita's GLR parsing algorithm [37] quickly. Earley's and GLR would allow any kind of grammars – not just LL(1) or LR(1) or LALR, but *any*. This was a unique and stimulating feature for a parser generator, since putting together rules from different modules would also work (i.e. parsing would happen) and often it would do the right thing. SDF with GLR as its underlying execution mechanism offered the powers of modularity, language embeddings, and compositionality of grammars that were eminently useful and also deemed elegant.

However, ambiguity and its cure disambiguation are neither modular nor compositional. By this, we mean that two arbitrary unambiguous modules when composed can easily become ambiguous and that two arbitrary modules with disambiguation constructs without parsing errors could when composed, easily generate spurious parsing errors. Ambiguity of context-free grammars is generally undecidable (with and without disambiguation constructs) [36], and this conundrum is the main motivation for all our research into the "diagnostics and treatment" of ambiguity in the SDF world [43]. Ambiguity made the SDF sometimes hard to use, despite its elegance and compositionality, or perhaps because of it. On top of this, the non-determinism of SDF grammars (ambiguous or not) is also a source of inefficiency of Tomita's GLR. The same disambiguation constructs that were proposed would also have a positive effect on efficiency as well.

Rewinding, this history starts with the implementation of SDF which was documented in Jan Rekers' thesis on incremental general parser generation in 1992. SDF existed before this and was documented in the technical reports of the ESPRIT project "GIPE - Generating Interactive Programming Environments" and GIPE II in ESPRIT 2 ([18] not digitized). SDF and the initial implementations of SDF2 were implemented using ASF+SDF itself, which was implemented in "LeLisp" [15]. The scanner in SDF was non-deterministic as well: scanning would produce all possible tokens at a given input position, from which the non-deterministic parsers could choose. This was necessary to achieve the desired modularity and compositionality of real programming languages like PL/I, Pascal, and COBOL. Disambiguation was featured in SDF by the associativity and priority declarations between rules, to help declare the binding strength of unary, binary, and n-ary expression operators without having to factor a grammar and without introducing any helper non-terminals. Writing and maintaining SDF grammars was attractive because due to these two disambiguation constructs the number of rules needed to define the syntax of a language was kept close to the number of actual constructs in the language. The interactions between the scanner and parser were sometimes unpredictable due to complex feature interactions with language composition and rules like longest match and keyword reservation. Incremental parsing was important at that time for feasibility of running complex and lively updated IDEs on small and slow machines. Wilco Koorn and Jan Rekers extended the GLR algorithm for substring parsing [32] to that end, and also to improve error recovery and auto-completion IDE features. This pushed the interaction between the parsing and the scanning algorithms to the limit.

Eelco's Ph.D. thesis on SDF2 in 1997 followed [45], and we completed an implementation of its parsing and disambiguation mechanisms (SGLR and PGEN) in C and ASF+SDF in 2000 together with Jeroen Scheerder and Mark van den Brand [7]. A main driving force at that time was the COBOL grammar by Chris Verhoef and Ralf Lämmel [27], as well as the "Island Grammars" by Leon Moonen [29] that kept pushing the boundaries of what was possible with declarative disambiguation. Peter Mosses and the Action Notation [9] (design and implementation) we considered important to satisfy as our "customer on-site." Eelco had been inspired by Solomon and Cormack [33] and introduced "scannerless parsing" to SDF by removing the entire scanner from the architecture, thereby introducing lexical ambiguity to the playground of SDF2. This design decision removed the aforementioned hard-to-predict interactions between a non-deterministic scanner and a non-deterministic parser.

Eelco's thesis contains the mitigations necessary to make scannerless parsing workable. These are two new disambiguation constructs: *follow restrictions* for longest match and *reject rules* for keyword reservation [39]. He also introduced a simplified representation of parse trees, with only three kinds of nodes: applications of grammar rules, unordered ambiguity clusters, and terminal characters. The grammar rules were represented in the abstract syntax of SDF2 as an algebraic data type in the ATerm format (Pieter Olivier, Hayco de Jong, Mark van den Brand, Paul Klint) [11, 10]. This algebraic format of constructors for parse trees, called AsFix2, was derived from earlier parse tree export formats (AsFix1) that were designed by Mark van den Brand to bootstrap ASF+SDF off of the LeLisp implementation [38] and as the intended exchange format between the various tools of the ASF+SDF Meta-Environment [7] that was under development at that time.

Since SDF2 was partly implemented in ASF+SDF itself, a bootstrap was required. Mark van den Brand and Pieter Olivier completed the LeLisp-independent ASF+SDF compiler in ASF+SDF in 2001, which included a compilation of the implementation of SDF2 and a re-implementation of the back-end table generator in C. My own thesis work included disambiguation for SDF2 [42]. That was in collaboration with Eelco, Jeroen Scheerder, and Mark van den Brand [39], on how to implement the filtering ideas in Eelco's thesis [26, 45]. The work with Rob Economopoulos was about integrating RNGLR [34] into SGLR in 2013 [17], which could simplify some of the filters but complicated others. Diagnosing ambiguity with Bas Basten was a parallel track [43, 5] (2011).

SDF2 as-is was used for many years by the ASF+SDF Meta-Environment, ELAN4 environment [12, 41], Action Environment [9] and StrategoXT [13] communities. In this essay we focus on the differences between the bottom-up parsing architecture of SDF2 as it was in the early 2010's and the Rascal top-down architecture in the same period.

SDF2's Scannerless Generalized LR parsing algorithm (SGLR) is by Eelco Visser [45]. What makes it different from its predecessors, namely Rekers' fixed version [31] of Tomita's GLR [37], is the semantics or implementation of disambiguation filters specific to scannerless parsing and specific to declarative definitions of operator precedence and associativity in expression grammars. SGLR is the parsing architecture made ready to implement SDF2, the syntax definition formalism from the thesis of Eelco Visser. The predecessor of SDF2, SDF had similar disambiguation constructs for operator precedence, but not for disambiguating lexical syntax. Hence Eelco named "SGLR": "Scannerless GLR", and the efficient and declarative disambiguation of lexical syntax is its core contribution.

Lexical ambiguity aside, what we never really solved (at that time) was the context-free ambiguity problem in general. Context-free general parsing produces multiple parse trees, sometimes, and it is hard to predict when. And, SGLR offers specific disambiguation constructs for specific kinds of ambiguity [5], but it can not solve arbitrary ambiguity. So, we experimented (from the very start) with far more general disambiguation constructs, such as the "multiset filter" algorithm [26, 45]. The multiset filter lifts the strict partial order of priority rules to entire parse trees by considering them "sets of rules" and applying a strict partial order of sets of partially ordered elements on entire trees. The more advanced the filters became, the less declarative and the more heuristic they became. Also, new disambiguation filters tended to be hard to implement correctly (more on this later) or efficiently (they all became back-end tree filters). Every new disambiguation concept added to SGLR required almost the effort of a PhD thesis. At the end of the 2010's, we found a resting point, eventually, to be able to filter parse trees using general purposes tree manipulators, such as Stratego, ASF+SDF, and Rascal – a.k.a. semantics directed disambiguation [8].

The second bootstrap stage of Rascal in 2011 provided us with a choice again. We had the opportunity to start from scratch in terms of parsing architecture. Having wrestled with the GLR algorithm for decades, we decided to flip the perspective and go top-down to Scott and Johnstone's GLL [35]. The one and only motivation was the simplicity and elegance of the new architecture, promising also simple and elegant disambiguation filters. A scannerless version of GLL [35] with disambiguations based on all the filters of SDF2 was indeed produced for Rascal 0.4.x. After this experimenting with new filters became really easy, intuitive, and fast. It is the goal of the current essay to substantiate this story.

The PhD thesis of Izmaylova and Afroozeh contains the idea of mapping the semantics of disambiguation filters to (lexical) constraints in data-dependent context-free grammars [1]. The team of Jim, Mandelbaum, and Walker had shown with their Jakker parser generator that data-dependent grammars are an elegant formalism for expressing the unambiguous syntax of programming languages [20, 21]. Moreover, such data-dependent grammars with lexical constraints seem to effectively model almost every hack we have seen in hand-written top-down parsers as well. Even the offside rule in Haskell, which is described as a hack of introducing an extra token in the token stream in an error state, can be simulated using a DDCFG in Iguana [2], and also symbol tables such as used in the scanning of C programs are expressible in data-dependent context-free grammars [1].

2 Comparing Syntax Definition Formalisms

These are the three main disambiguation constructs in SDF2 and Rascal:

- **Priorities and associativity** for binding strength of operators in expression languages;
- **Reject rules** for keyword reservation;
- **Follow restrictions** for longest and first match.

```
module ExpInSDF2
context-free syntax
  Id          -> Exp
  "(" Exp ")" -> Exp {bracket}

context-free priorities
  Exp "[" Exp "]" -> Exp <0> >
  Exp "*" Exp     -> Exp {left} >
  { left:
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp }
```

```
module ExpInRascal
syntax Exp
  = Id
  | bracket "(" Exp ")"
  | Exp "[" Exp "]"
  > left Exp "*" Exp
  > left ( Exp "+" Exp
         | Exp "-" Exp
         );
_
_
```

Figure 1 Comparing expression language definition between SDF2 and Rascal; minor meta-syntax differences but conceptually the same.

2.1 Associativity and Priority Disambiguation

In Figure 1 the use of priority and associativity declarations is shown for the same language "Exp" in both SDF2 and Rascal. Associativity can be applied to a single rule, e.g. $*$, or a group of rules ($-$ and $+$). We see a binary ordering $>$ which is to be interpreted as a strict partial order (transitive, irreflexive, asymmetric) between rules.

In both formalisms, each priority and associativity declaration defines a set of disallowed derivation steps in the respective grammar. Namely for "left" associative binary recursive rules a rule in the same group must not be derived on the right-hand side non-terminal of the rule. Vice versa for "right" associativity. If a rule is not binary recursive (i.e. it is of the form `X ::= X ... X` then the filter has *no effect.*

Also, both formalisms share similar semantics for the priority relation. If production A has a higher priority than another production B, then never shall B be a direct child of A. No non-terminal of A will be recognized using the application of rule B. SDF2 priority rules have no effect unless the two rules in question are shaped like so: `X ::=` ϵ `X ... | ... X` ϵ or so: `X ::= ... X` ϵ `|` ϵ `X ...`. The recursive positions must overlap at a left-most or right-most position (or their pre/postfixes must derive the empty sequence ϵ).

For SDF2, if the user makes a mistake and defines a priority relation that is reflexive or symmetric ($A > B$ and $B > A$), then the filter may remove too many derivations from the generated parser with parse errors as a result. Also, SDF2 removes *all* nested derivations of B under A if $A > B$, including the position between brackets in `Exp "[" Exp "]"`. This would make it impossible to write `a[1+2]`. And so SDF2 users write $< 0 >$ before that rule to limit the filter to the first argument position and ignore it for the second. For Rascal the set of generated filter positions is filtered itself; only the positions that are guaranteed to generate ambiguity are filtered and the other positions are ignored. If the priority relation is not a partial order, the user is provided with an error message.

In short: the semantics of associativity and priority disambiguation is described in terms of derivation step filters on the grammar level. We refer to Eelco's thesis [45], this paper on ambiguity diagnostics [5], and this on disambiguating expression grammars [3], which explain the above formally.

The correctness of the semantics of these constructs relies on the guaranteed ambiguity of binary expression operators that are left and/or right-recursive, such that the filter does not remove the last derivation from a Parse Tree [3]. So, their implementation should prevent the effects of such derivations or remove them, somewhere in the respective parsing architecture in order to implement this filtering behavior.

```
module RejectInSDF2
context-free syntax
  Id           -> Exp
  Keyword      -> Id {reject}
  "begin"      -> Keyword
  "end"        -> Keyword
```

```
module RejectInRascal
syntax Exp = Id \ Keywords;
keyword Keywords
  = "begin"
  | "end"
  ;
```

Figure 2 Comparing reject rules between SDF2 and Rascal.

```
module RestrictionsInSDF2
lexical syntax
  [A-Za-z][A-Za-z0-9]* -> Id
lexical restrictions
  Id -/- [A-Za-z0-9]
```

```
module RestrictionsInRascal
lexical Id
  = [A-Za-z][A-Za-z0-9]*
  !>> [A-Za-z0-9];
_
```

Figure 3 Comparing follow restriction between SDF2 and Rascal.

2.2 Reject Rules

Figure 2 depicts the two styles of reserving keywords as a disambiguation mechanism. The `reject` tag was introduced by Eelco Visser in 1997. The semantics is that any subsentence recognized by *any* derivation for `Id` which can also be recognized as `Keyword`, at that same input position and having the same length, must be filtered. As described, the mechanism extends context-free grammars to include the *intersection* of context-free non-terminals and it should be possible to generate parsers for the famous non-context-free example $a^n b^n c^n$. Later we read why this was not accomplished with SDF2.

The Rascal reflection of this design is the \ operator that removes the language of `Keywords` from *that specific* use of `Id` in `Exp`. The difference is thus that Rascal defined a derivation step filter for a specific position of `Id` in a specific rule, while SDF2 defined a filter for all uses of `Id` in any rule. Nevertheless, the expressive power would be the same, if the Rascal designers had not limited the keyword non-terminals to generate non-empty and finite languages only.

2.3 Follow restrictions

Here in Figure 3 we even more clearly step out of the realm of context-free grammars. Follow restrictions in Rascal and SDF2 express, literally, constraints on what comes *after* the character yield of a recognized non-terminal. In SDF2 we define a filter that removes all derivations of `Id` anywhere if the single character that would follow it in the input is a member of the character class `[A-Za-z0-9]`. Although this is not a local property of a Parse Tree, it is a local input of most parsing algorithms that move from left to right through the input. The next character or token in the stream is usually referred to as the "lookahead token." SDF2 also has multi-character lookahead tokens for restrictions, which lead to the same semantics (but an arguably much more complex implementation).

In Rascal the semantics is similar, but we define the derivation filter not for all instances of the non-terminal, but only for the position in the rule that the restriction is applied. Next to this Rascal features the complement: a follow requirement declares that a non-terminal *must* be followed by a certain character class, and their analogous duals: precede restrictions and requirements. Rascal, like SDF2, also features multiple look-ahead characters.

While explaining the semantics of the three disambiguation constructs we used only ideas such as Grammar, Derivation steps, Parse Trees, and Input sentences. There is no distinction between top-down and bottom-up because we have yet to dive into the implementations of these filters.

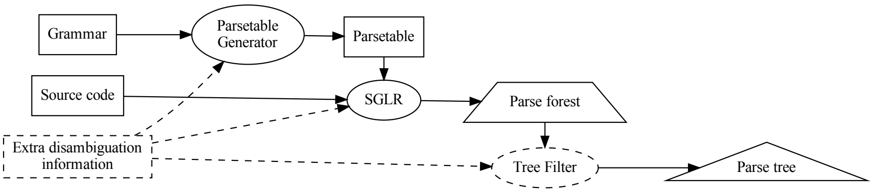


Figure 4 Bottom-up SDF2 architecture based on SGLR.

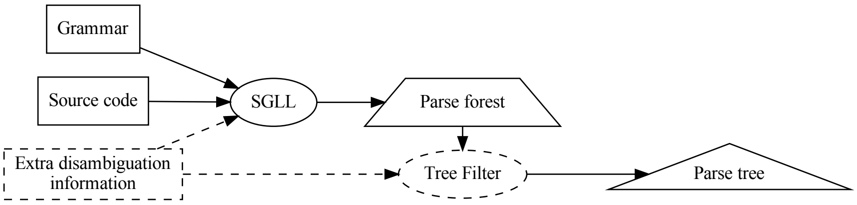


Figure 5 Top-down Rascal architecture based on SGLL.

3 Comparing Parsing and Disambiguation Algorithms

We are comparing the *parsing and disambiguation* algorithms SGLR and SGLL as they were part of the parsing architectures for SDF2 and Rascal as they were in 2010. We will show details of the SGLR implementation in C and ASF+SDF and in Java and Rascal of the Rascal implementation.

Figure 4 shows the parsing and disambiguation architecture of SDF2 with SGLR in it. As you can see a disambiguation filter may end up filtering a rule from a grammar completely, modify the SLR parse table to prevent certain derivations to occur at all, or inject itself in the parser run-time to prevent parser driver operations that have the same effect. Eventually, every filter could be implemented by a post-parse tree transformation.

Figure 5 shows a different architecture because there is no intermediate stage for parse table generation. However, Rascal does have a grammar rule merging step while loading a new grammar that captures some of the partial evaluation that is done while building parse tables as well.

The two premises of implementing SDF2's disambiguation constructs are [26, 39]

- efficiency: implement filters as early as possible in the parsing architecture; preventing non-determinism is better than fixing ambiguity later. For Rascal, we adopted a similar but less far-fetching dogma: better filter while predicting than filter while accepting a derivation.
- grammar neutrality: do not change the shape of the grammar rules, such that also the shape of the Parse Trees is unaffected. This adds to the predictability of the shape of the forests as well as efficiency since tree structure does not need to be reconstructed. For Rascal, this is also a core design constraint.

3.1 Implementing priority and associativity

In a bottom-up parser, it is possible to *completely prevent* the creation of derivations that do not satisfy the constraints generated from associativity and priority declarations[1] The major vehicle for this is the parse table generator.

SDF2 uses DeRemer's SLR(1) [14] table construction. An SLR(1) parse table is something most students must be able to generate from a grammar by hand in their Compiler Construction course. The table represents a state machine for a pushdown automaton driver that will recognize a string or not, using the information in the table. One could look at the table as a partially evaluated parsing algorithm (typically Earley's algorithm [16]) where the grammar is interpreted but the input sentence is left open. Eelco's idea is that any **reduce** P_1 actions going out of a state can be completely removed if said state witnesses that P_2 is its parent at the wrong position according to $P_2 > P_1$.

However, SLR(1)'s follow sets are defined on non-terminals and not production rules, so they do not represent rules that clearly. Every goto action out of a state may represent a union of rules for different non-terminals and different positions in different rules of the same non-terminal. Seeing that we use a non-deterministic parsing driver, Eelco observed that it was possible to redefine SLR follow sets per production instead of per non-terminal; and this enabled a full implementation of the priority and associativity semantics. The overhead is that different rules would exit a state on sometimes the same follow set to the same state, but at least never could an illegal transition be made anymore. The breakthrough here was that this solution, on top of Rekers' version of Tomita's GLR, could deal with *any* context-free grammar and not only the LR(k) class.

The actual code that implements this filter in SDF2 is written in ASF+SDF and C (against ApiGen-generated ATerm interfaces). The expensive part is the transitive closures and cartesian products part, which was ported to C after the declarative version in ASF+SDF proved to be a bottleneck. This implementation derives triples (P_1, pos, P_2) that explain for every rule P_1 at which position *pos* the other rule P_2 should be disallowed. The worst-case size of the set of these triples is quadratic in the number of original production rules. Later the relatively simple SLR table generator takes this set as an additional argument and surgically does not add reductions if they occur in the set (Figure 6).

The Rascal implementation of the same filter uses a comparable triplet computation but is written in (higher-level) Rascal. The Rascal code also first proves ambiguity for left-most and right-most recursive positions, instead of applying the filter to all arguments of every production. With SGLL, the set of triplets is not used at parser generation time but during the prediction stages of the top-down parsing algorithm (Figure 7). Rascal's SGLL (designed and implemented by Arnold Lankamp) filters rules the moment they are applied by looking at their parent node. At this moment of rule reduction, the triplet set is queried. This has the exact same effect as the SDF2 implementation, at the cost of a hash-table lookup. In fact, the SGLL implementation numbers every production with a low integer and uses a small array to look up all the conflicting rules at a certain position. In a correct SGLL or SGLR implementation, all these solutions for priority and associativity filtering have the same effect of removing reductions without breaking the algorithm, at the cost of extra bookkeeping.

The SGLL algorithm prevents certain recursive steps dynamically while the SDF2 algorithm filters certain derivations. From a slight distance, both algorithms simulate a grammar transformation that would introduce non-terminals for every production rule and

[1] Later Peter Mosses found out that there are cases where SDF2 is theoretically incomplete for expressing binding strength using single derivation step filters and that required the development of a new theory and new implementations. This is out of the scope of the current paper.

```
 1  ATermList shift_prod(ItemSet items, int prodNr) {
 2    Item item, newitem;
 3    PT_Symbol symbol;
 4    ATermList newvertex = ATempty;
 5    ItemSetIterator iter;
 6    PT_Production prod = PGEN_getProductionOfProdNumber(prodNr);
 7
 8    symbol = PT_getProductionRhs(prod);
 9
10    ITS_iteratorPerDotSym(items, symbol, &iter);
11    while (ITS_hasNext(&iter)) {
12      item = ITS_next(&iter);
13      assert(PT_isEqualSymbol(symbol, IT_getDotSymbol(item)));
14      newitem = IT_shiftDot(item);
15      if (newitem != NO_ITEM
16          && !PGEN_isPriorityConflict(item, prodNr)) {
17        newvertex = ATinsert(newvertex, IT_ItemToTerm(newitem));
18      }  }
19
20    return newvertex; }
```

Figure 6 C code snippet in the SDF2 parser generator with a single surgical addition predicate on line 16 to implement associativity/priority filtering. Small intervention: big impact.

disallow certain rules based on the same constraint triplets. A factored grammar, such as we see when using LALR parsers, would probably not be much different. One could say that by requiring not to change the grammar, we have to simulate those grammar changes on a *lower level of abstraction* to achieve the same effect. Accidentally, a similar "set of integer production rules representation" used by Arnold Lankamp in SGLL, Eelco had envisioned earlier with "character-class grammars" [44]. There each non-terminal would also be represented by a set of active production rules for that level in the grammar and removing a production would entail implementing a filter on the grammar level.

The SDF2/SGLR solution requires theory: does the implementation satisfy all the constraints derived from the semantics of the formalism? Well, only if we change the concept of what an SLR(1) table is a bit, such that it fits. The solution does not generalize to other standard table formats, like LALR(1). The Rascal/SGLL solution sits very tightly on the concept of top-down parsing where recursion on the way down models prediction and coming back up models acceptance/reduction; it can be added to any (G)LL(1) algorithm implementation technique.

3.2 Implementing reject

The way we write SDF2 `reject` rules already leaks something about the implementation strategy of keyword reservation. We simply schedule the rejected rule along with the rest of the grammar. Then, when *all alternatives* for the `Id` non-terminal have finished, we check if one of them was accidentally a `reject` rule and if so we remove all those derivations from the computation.

With this elegant approach, Eelco had found a near-optimal solution [45]. We do not have to start a whole other parser with its own stack. Instead, we surf on the non-deterministic graph-structured stack of Tomita's algorithm and pay only a low overhead of recognizing an additional alternative.

However, this implementation had to be revisited and revised many times between the years 2000 and 2005, and eventually, we had to admit general non-terminals could not be "rejected" by this algorithm. The problem with `reject` is very much akin to the original bug

```
 1  private void handleEdgeListWithRestrictions(...) {
 2    firstTimeRegistration.clear();  firstTimeReductions.clear();
 3    for (int j = edgeSet.size() - 1; j >= 0; --j) {
 4      AbstractStackNode<P> edge = edgeSet.get(j);
 5      int resultStoreId = getResultStoreId(edge.getId());
 6
 7      if (!firstTimeReductions.contains(resultStoreId)) {
 8        if (firstTimeRegistration.contains(resultStoreId)) continue;
 9
10        firstTimeRegistration.add(resultStoreId);
11
12        if (!filteredParents.contains(edge.getId())) {
13          AbstractContainerNode<P> resultStore = null;
14          if (edgeSet.getLastVisitedLevel(resultStoreId) == loc)
15            resultStore = edgeSet.getLastResult(resultStoreId);
16          ... /* elided error recovery code */
17          resultStore.addAlternative(production, resultLink);
18        } else {
19          AbstractContainerNode<P> resultStore = edgeSet.getLastResult(resultStoreId);
20          stacksWithNonTerminalsToReduce.push(edge, resultStore);
21        } } } }
```

Figure 7 Java code snippet in Rascal's SGLL parser run-time, with a single additional predicate on line 14 to filter associativity/priority violations. Again: small intervention; big impact.

```
public interface ICompletionFilter {
  boolean isFiltered(int[] input, int start, int end, PositionStore positionStore); }
```

Figure 8 Rascal's SGLL completion filter interface code.

in Tomita's algorithm, which occurred with hidden left recursion. Sometimes the graph-structured stack would miss reductions and Farshi fixed that [30] with an additional stage to search for the missing reductions. The `reject` implementation very much depends on the algorithm identifying a moment where all rules for the restricted non-terminal `Id` have reduced, but the original algorithm for it did not achieve this. The reject "aspect" of SGLR is scattered in different places making it hard to theorize what its effect really is. And so sometimes rules would continue even though they should have been rejected with spurious ambiguity as a result. After several experiments, the diagnosis was left-nullable rules for the restricting could lead to "escaped" reductions. Further complicating the algorithm with yet more searching on top of Farshi's fix was deemed inefficient.

The reject filter in Rascal is either an implementation of an `ICompletionFilter` or an `IEnterFilter`. The latter prevents going into a production when it is predicted while the former removes the effect of recognizing a production when it is completed. This is the general filtering scheme, which gives access to the input character array, the start, and end index of the currently recognized input, etc. Other (static) information, like which non-terminal or production is captured by the object that implements the filter. The reject filter in SGLL simply compares the input subsentence with a given list of keywords that are not allowed and fails on a match. A more complex implementation could parse the substring using another parser (or recognizer). Setting up a nested parser is not as complex or expensive in Java as it was in SGLR's C version.

The parameters of a completion filter in SGLL (see Figure 8) make explicit what can be safely used as information to filter without breaking the algorithm's assumptions. The dynamic programming techniques that are used to stay in polynomial time for an exponential number of parse trees, or even cubic, are predicated upon the identification of reusable parse stacks for reusable subsentences. When context information breaks into this equation, it breaks the underlying assumptions for sharing computations leading to false positives (spurious derivations) as well as false negatives (spurious filtering of derivations). Every new filter introduced requires new theory. The definition of `ICompletionFilter` and `IEnterFilter` in Rascal's SGLL mitigate this by allowing any filter based on the given information and guaranteeing algorithmic correctness. If you need something more, it's back to the drawing board just as with SGLR.

3.3 Implementing follow restrictions

The final filter, however innocuous, proved to be another grand challenge for implementing in SDF2 and SGLR. Firstly, the basic implementation for single-character lookahead was simply to remove the given characters from the lookahead sets in the SLR(1) table [45]. Secondly, multiple character lookahead would be implemented by dynamically filtering reductions in the inner parser loop. With the right internal administration that associates the lookaheads with every production rule, the code change in the algorithm is minimal.

However, at the time we were using the ATerm library for representing parse trees with its *maximal sharing* ability. Parse tree nodes that were structurally equal, would always be shared. Sharing was thus based on the contents of the trees' sub-nodes, and not on the context. This design decision is efficient in the context of ambiguity where lots of sub-nodes would be structurally equal (whitespace). With a theory of context-free grammars this all works fine. With the theory of context information with follow restrictions, maximal sharing breaks the parser. To fix the bugs, we ended up introducing an additional lookup table for ambiguous parse trees where the starting position in the input sentence became part of the lookup key.

Having learned from the experience, the Rascal SGLL implementation used its own intermediate SPPF-like [37] data structure for parse forests, which is then serialized to AsFix Parse Trees after the parse is done. The aforementioned `ICompletionFilter` can be used to implement follow restrictions in a single line of code. `IEnterFilter` would be used for the precede variants. If you'd start from scratch to implement SGLR in Java, for example, you would use the same trick. Indeed the JSGLR code by Karl Trygve Kalleberg in Spoofax, uses an ambiguity hash-table that also includes the start position of every tree.

3.4 Top-down disambiguation is easier to get right

Table 1 summarizes how the three declarative implementation constructs were implemented in either a top-down (Rascal) or bottom-up (SDF2) parsing architecture. Once you have the theory straight, the implementation of a disambiguation filter seems a surgical incision in either algorithm: a conditional around the scheduling of the next step. If only this were true.

Implementing the Reject and Follow Restriction filters has proven to be complex for the SGLR architecture while it is straightforward in SGLL. Moreover, the priorities and associativity filter on the parse table level can never be complete, even though it is elegant. Small changes in the parse table, such as filtering a goto edge, may break the conditions under which parse stacks or parse tree nodes can be shared later in the GLR algorithm. These semantic links are platonic, in the sense that they do not lead to explicit dependencies on the source code level of SGLR, however, when not taken care of complex bugs do arise. Concretely, the addition of Follow Restrictions to the SLR table construction algorithm broke many of the underlying assumptions of the SGLR implementation and required the reconsideration of all of its internal data structures. On the other hand, for SGLL a follow restriction was a simple conditional while scheduling the next algorithmic step.

Table 1 Overview of the disambiguation implementations for either SDF2 (bottom-up) or Rascal (top-down).

Feature	Grammar	Implementation
SDF2 Priority / Associativity	$P_1 > P_2$, `left`, `right`	1. Compute constraint triplets 2. SLR(1) follow-sets per rule 3. Filtering goto's in SLR(1)
Rascal Priority / Associativity	$P_1 > P_2$, `left`, `right`	1. Compute constraint triplets 2. Filter forest edge creation
SDF2 Keyword reservation	`Kw -> Id {reject}`	1. Grouping reductions 2. Filter reductions
Rascal Keyword reservation	`Id \ Kw`	1. `ICompletionFilter`
SDF2 Longest Match	`Id -/- [A-Z]`	1. Goto Filter on follow sets 2. Reduction filter for k lookahead
Rascal Longest Match	`Id !>> [A-Z]`	1. `ICompletionFilter`

The reject filter for SGLR proved to be even more difficult to implement. The actual filter operation is to remove all other reductions for a non-terminal in the presence of a "rejected" one for the same sub-sentence and non-terminal. The SGLR algorithm does not schedule reductions in such a way that a clear moment arises when all possible reductions for a subsentence have been collected. Sometimes graph stack nodes are processed already for further reductions (chain rules), and that way a tree would escape that would otherwise have been filtered. Sometimes the rejected stack node itself would be processed too early, letting later nodes escape. The first problem was solved by changing the GLR algorithm to group reductions in the same starting position of the input. The latter problem was solved by disallowing more complex non-terminals to be rejected, limiting them to finite non-nullable languages. The reject filter in SGLL is a simple reduction filter.

We conclude that top-down is much easier to experiment with and extend. Bottom-up could be faster due to partial evaluation, but still, additional bookkeeping and less sharing are required to filter correctly. Rascal's SGLL in Java is as fast as the SDF2's SGLR in C.

4 Perspective on contextual disambiguation with DDCFGs

Let's step back from the comparison made between top-down general and bottom-up general parsing with disambiguation and zoom out to the general problem of disambiguation.

- All three disambiguation constructs use *context information.*
- Each of the three disambiguation constructs is an *ad-hoc* extension of the SDF [43, 6].
- Each disambiguation construct deeply impacts the parsing algorithm.

Never mind that they are easier to implement in GLL, but what is the best way of formulating the next disambiguation construct on the SDF level? Say we want to support the offside rule [28]. How to implement it in (S)GLL? Disambiguation always adds "context" to the algorithm of constructing parse trees as compared to "context-free" grammars. At the LDTA conference in 2011, Trevor Jim and Yithzak Mandelbaum demonstrated the utility and elegance of data-dependent context-free grammars with their Yakker parser generator [21, 20]. Much earlier, Mark van den Brand in his PhD thesis [40] also demonstrated that parse-time semantic predicates can be used elegantly and efficiently to disambiguate (lexical and context-free) ambiguity.

A Data-dependent Constraint Grammar is a context-free grammar with three major extensions: (a) the non-terminal on the left-hand side of any rule may receive additional data parameters, (b) every symbol on the right-hand side may be conditional on said data parameters using constraint formulas, and (c) data from the input sentence or of syntax trees already processed may be passed as parameters to non-terminals or constraints. Typical "data" would be the character string of a sub-sentence, the start and ending position of every rule, the current indentation level, etc. Typical formulas would be integer arithmetic (for layout positioning), and string (in)equality, but in general, any predicate *without side-effects* written in the host programming language is ok.

Afroozeh and Izmaylova [2, 1] mapped all of SDF's and Rascal's disambiguation mechanisms to their Iguana formalism which is based on data-dependent grammars, and then immediately added many more disambiguation constructs. For example, with Iguana it is possible to declaratively express the offside rule and many other "two-dimensional" layout constraints for programming languages such as Haskell and Python. Iguana is a top-down parsing architecture, as the reader might expect. It is also possible to implement data-dependent grammars on top of Earley's algorithm [21, 16].

However, the semantics of Disambiguation remains the same whether you implement your DDCFG parsing algorithm in a top-down or a bottom-up data-dependent context-free general framework. The formal semantics of data-dependent context-free grammars acts as a virtual machine for disambiguation constructs, making it easier to reason about correctness independent of the implementation in a complex parsing algorithm [1].

5 Conclusion

First, contextual disambiguation is a pleonasm. Second, it is arguably easier to design and implement (new) disambiguation constructs with GLL than with GLR. Third, data-dependent context-free grammars add the level of formality and generality that we were always searching for when inventing new disambiguation schemes (as exemplified by the Jakker and Iguana Parsing Architectures). We conclude that a *top-down* implementation of *data-dependent* context-free parsing is the way to go for Rascal as well as SDF3.

References

1 Ali Afroozeh and Anastasia Izmaylova. One Parser to Rule Them All. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2015, pages 151–170. ACM, 2015. doi:10.1145/2814228.2814242.

2 Ali Afroozeh and Anastasia Izmaylova. Iguana: a practical data-dependent parsing framework. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 267–268. ACM, 2016. doi:10.1145/2892208.2892234.

3 Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen J. Vinju. Safe specification of operator precedence rules. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2013.

4 Luís Amorim and Eelco Visser. *Multi-purpose Syntax Definition with SDF3*, pages 1–23. Springer, September 2020. doi:10.1007/978-3-030-58768-0_1.

5 Bas Basten and Jurgen Vinju. Parse forest diagnostics with dr. ambiguity. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2011.

6 Bas Basten and Jurgen Vinju. Parse forest diagnostics with Dr. Ambiguity. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2011.

7 Mark G.J. van den Brand, Arie van Deursen, Jan Heering, Hayco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul. Klint, Leon Moonen, Pieter .A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

8 Mark G.J. van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized Parsing and Term Rewriting – Semantics Directed Disambiguation. In Barret Bryant and João Saraiva, editors, *Third Workshop on Language Descriptions Tools and Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, 2003.

9 Mark van den Brand, Jørgen Iversen, and Peter Mosses. An Action Environment. *Electr. Notes Theor. Comput. Sci.*, 110:149–168, December 2004.

10 Mark van den Brand and Paul Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information & Software Technology*, 49:55–64, January 2007.

11 Mark van den Brand, Paul Klint, Hayco de Jong, and Pieter Olivier. Efficient annotated terms. *Software— Practice & Experience*, 30(2), January 2000.

12 Mark van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Environments for term rewriting engines for free! In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, RTA'03, pages 424–435, Berlin, Heidelberg, 2003. Springer-Verlag.

13 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008. Special Issue on Second issue of experimental software and toolkits (EST). doi:10.1016/j.scico.2007.11.003.

14 Frank DeRemer. Simple lr(k) grammars. *Commun. ACM*, 14(7):453–460, 1971. doi:10.1145/362619.362625.

15 Arie Van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach: Vol. V.* World Scientific Publishing Co., Inc., USA, 1996.

16 Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13:94–102, February 1970. doi:10.1145/362007.362035.

17 Giorgios R. Economopoulos, Paul Klint, and Jurgen J. Vinju. Faster scannerless GLR parsing. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2009. doi:10.1007/978-3-642-00722-4_10.

18 J. Heering and P. Klint. A syntax definition formalism, 1986. ESPRIT"86: Results and Achievements, page 619–630.

19 Jan Heering, Paul R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF — reference manual. *SIGPLAN Not.*, 24(11):43–75, November 1989. doi:10.1145/71605.71607.

20 Trevor Jim and Yitzhak Mandelbaum. Delayed semantic actions in yakker. In Claus Brabrand and Eric Van Wyk, editors, *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, page 8. ACM, 2011. doi:10.1145/1988783.1988791.

21 Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and algorithms for data-dependent grammars. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 417–430. ACM, 2010. doi:10.1145/1706299.1706347.

22 Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *SIGPLAN Not.*, 45(10):444–463, October 2010.

23 Paul Klint. *From SPRING to SUMMER: design, definition and implementation of programming languages for string manipulation and pattern matching.* PhD thesis, Technische Hogeschool Eindhoven, March 1982.

24 Paul Klint, Tijs van der Storm, and J.J. Vinju. EASY meta-programming with Rascal. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *LNCS*, pages 222–289. Springer Berlin / Heidelberg, 2011.

25 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009. doi:10.1109/SCAM.2009.28.

26 Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

27 Ralf Lämmel and Chris Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

28 P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9:157–166, March 1966.

29 Leon Moonen. Generating robust parsers using island grammars. *Proceedings Eighth Working Conference on Reverse Engineering*, pages 13–22, 2001.

30 Rohman Nozohoor-Farshi. Handling of ill-designed grammars in tomita's parsing algorithm. In *Proceedings of the First International Workshop on Parsing Technologies*, pages 182–192, Pittsburgh, Pennsylvania, USA, August 1989. Carnegy Mellon University. URL: https://aclanthology.org/W89-0219.

31 J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

32 Jan Rekers and Wilco Koorn. Substring parsing for arbitrary context-free grammars. In *Proceedings of the Second International Workshop on Parsing Technologies*, pages 218–224, Cancun, Mexico, February 13-25 1991. Association for Computational Linguistics. URL: https://aclanthology.org/1991.iwpt-1.25.

33 D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI 1989, pages 170–178. ACM, 1989. doi:10.1145/73141.74833.

34 Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006.

35 Elizabeth Scott and Adrian Johnstone. GLL parsing. *ENTCS*, 253(7):177–189, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

36 Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

37 M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

38 Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: The ASF+SDF compiler. *CoRR*, cs.PL/0007008, 2000. URL: https://arxiv.org/abs/cs/0007008.

39 Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, volume 2304 of *LNCS*, pages 143–158. Springer, 2002.

40 Mark G. J. van den Brand. *PREGMATIC – A generator for incremental programming environments*. PhD thesis, Radboud University Nijmegen, 1992.

41 Mark G. J. van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen. The ELAN Environment: an Rewriting Logic Environment based on ASF+SDF Technology. In *Workshop on Language Descriptions, Tools and Applications – LDTA'02*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Colloque avec actes et comité de lecture. internationale. URL: https://hal.inria.fr/inria-00101028.

42 J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* PhD thesis, Universiteit van Amsterdam, November 2005.

43 Jurgen J. Vinju. SDF disambiguation medkit for programming languages. Technical Report SEN-1107, Centrum Wiskunde & Informatica, 2011. URL: `http://oai.cwi.nl/oai/asset/18080/18080D.pdf`.

44 Eelco Visser. From context-free grammars with priorities to character class grammars. In Mieke Brune Arie van Deursen and Jan Heering, editors, *Dat Is Dus Heel Interessant, Liber Amicorum dedicated to Paul Klint.* Centrum Wiskunde & Informatica and IVI Universiteit van Amsterdam, 1997.

45 Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, Universiteit van Amsterdam, 1997.

Scope Graphs: The Story so Far

Aron Zwaan
Delft University of Technology, Netherlands

Hendrik van Antwerpen
GitHub, Amsterdam, Netherlands

Abstract

Static name binding (i.e., associating references with appropriate declarations) is an essential aspect of programming languages. However, it is usually treated in an unprincipled manner, often leaving a gap between formalization and implementation. The scope graph formalism mitigates these deficiencies by providing a well-defined, first-class, language-parametric representation of name binding. Scope graphs serve as a foundation for deriving type checkers from declarative type system specifications, reasoning about type soundness, and implementing editor services and refactorings. In this paper we present an overview of scope graphs, and, using examples, show how the ideas and notation of the formalism have evolved. We also briefly discuss follow-up research beyond type checking, and evaluate the formalism.

2012 ACM Subject Classification Theory of computation → Program semantics; Theory of computation → Program specifications; Theory of computation → Program analysis

Keywords and phrases scope graph, name binding, reference resolution, type system, static semantics

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.32

Acknowledgements We thank Casper Bach Poulsen, Douglas A. Creager, and the anonymous reviewers for their helpful comments to improve this paper.

1 Introduction

Formal presentations of type systems often abstract over surface language features. For example, names are assumed to be unique, or module systems are omitted. Although this yields concise and elegant calculi, it does not cover all the concerns real-world language implementations, such as compilers, interpreters and editors, have to deal with. Part of Eelco Visser's legacy is his work on principled and comprehensive approaches to specify name binding that do support those real-world programming language features. In his vision, language implementations should be derived from high level specifications expressed in *meta-languages* [23, 24].

Visser's meta-language research is reflected in the development of the Spoofax Language Workbench [7]. Initially, name binding and type checking were expressed in an imperative style, using the Stratego term rewriting language [2, 6]. The introduction of the NaBL (pronounced *enable*) language [8] was the first step towards *declarative* name binding specification. NaBL supports defining name binding rules for AST patterns for a whole range of common binding constructs, such as namespaced declarations and references, lexically nested scopes, transitive and non-transitive imports opening in the surrounding or subsequent scope, and type-dependent name resolution with overloading support. From these rules, an incremental name resolution algorithm was automatically generated [8, 25]. However, the semantics of NaBL were implementation-defined, documented by examples [10]. Attempts to define a declarative semantics for NaBL failed. Visser wrote: "[We] had a hard time explaining exactly how it worked. We kept getting the question 'but what is the semantics of NaBL?' and we didn't have a good answer. Our attempts at formulating a high-level concise formalization of the semantics underlying the implementation proved that this was a non-trivial problem." [22]

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 32; pp. 32:1–32:13
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The resulting research, discussed in this paper, was guided by the goal to design a meta-language for name binding that (i) has a clear and clean underlying theory (*principled*); (ii) can handle a broad range of common language features (*expressive*); (iii) is *declarative*, but realizable by practical algorithms and tools (*executable*); (iv) is factored into language-specific and language-independent parts, to maximize re-use (*reusable*); and (v) can be applied to erroneous programs as well as to correct ones (*resilient*) [17].

Traditionally, type systems deal with *names* using *environments*: finite mappings from names to types [13]. A *declaration* extends the environment, which is then used to type check the part of the program in which the declaration is in scope. A *reference* is type checked by looking up its name in the environment. This process of associating references with declarations is called *name binding*[1]. Environments work well for lexical binding such as lambdas, where names are scoped in subtrees of the binding construct, because they closely follow the structure of the abstract syntax tree (AST). However, it is more tedious to encode non-lexical binding such as imports or type members, where names are visible in parts of the AST that are not descendants or close siblings of the binding construct, this way. Therefore, environment-based type system formulations do not meet the goals set out before. Non-lexical binding feature are encoded in either a high-level style that is hard to translate to an executable type checker (as exemplified by the effort taken to formalize Scala's type system [1]), or in a more low-level style that is hard to reason with (cf. Hedin [5] for a similar observation about canonical attribute grammars). In both styles, the encoding is often very specific to the object language and difficult to reuse (cf. Pierce [13, ch. 15, 19, 20, 23, and 24], which all introduce custom encodings for particular new features). As a result, type checker implementations use other techniques, such as multiple passes over the program, to stage construction and use of environments [16, sect. 2.3]. In the end, the formalization and implementation differ in a way that makes it challenging to co-develop them, or verify their correspondence [3].

Luckily, although the details of name binding semantics differ across languages, there is a significant commonality below the surface. Recurring concepts are *scopes*, the "regions that behave uniformly with respect to name resolution" [10], *namespaces*, which categorize names based on their position in the program, and *shadowing*, strategies to disambiguate between multiple candidate declarations. Scope graphs [10] capture this uniformity using a reusable representation of name binding structure. In scope graphs, nodes represent scopes and declarations, which are connected by labeled edges. References are resolved by finding paths to eligible declarations, subject to visibility and shadowing policies expressed in terms of edge labels. Using this formalism, many different (non-lexical) binding patterns can be encoded.

Scope graphs have been embedded in type system specification meta-languages such as NaBL2 [17] and Statix [18, 16], implemented as part of the Spoofax language workbench [7]. Type checkers can be derived automatically from declarative type system specifications written in these languages. Type checker execution is performed by language-parametric algorithms that take care of operational aspects such as scheduling name lookups and scope graph construction, and are guaranteed to be sound with respect to the specification.

This paper gives an overview of published work related to scope graphs to date (visualized in Figure 1). First, we provide an introduction for readers without much background on scope graphs (Section 2). Next, we give an overview of all approaches that use scope graphs to derive sound type checkers from type system specifications, and the way the scope graph formalism has evolved (Section 3). Furthermore, we discuss how scope graphs are used

[1] We consider name binding to be *static* name binding, unless noted otherwise.

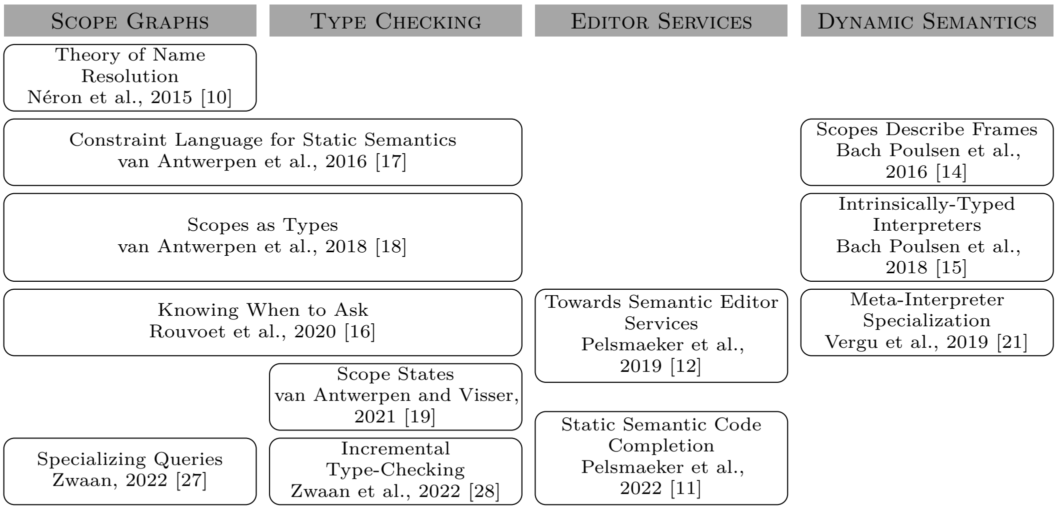

Scope Graphs
Type Checking
Editor Services
Dynamic Semantics
Theory of Name Resolution Néron et al., 2015 [10]
Constraint Language for Static Semantics van Antwerpen et al., 2016 [17]
Scopes Describe Frames Bach Poulsen et al., 2016 [14]
Scopes as Types van Antwerpen et al., 2018 [18]
Intrinsically-Typed Interpreters Bach Poulsen et al., 2018 [15]
Knowing When to Ask Rouvoet et al., 2020 [16]
Towards Semantic Editor Services Pelsmaeker et al., 2019 [12]
Meta-Interpreter Specialization Vergu et al., 2019 [21]
Scope States van Antwerpen and Visser, 2021 [19]
Static Semantic Code Completion Pelsmaeker et al., 2022 [11]
Specializing Queries Zwaan, 2022 [27]
Incremental Type-Checking Zwaan et al., 2022 [28]

Figure 1 Overview of publications related to scope graphs.

beyond type checking for editor services and reasoning about type soundness (Section 4). Finally, we reflect on the development of the formalism (Section 5). This paper is necessarily brief, but we hope it can serve as an introduction and starting point to learn more about scope graphs.

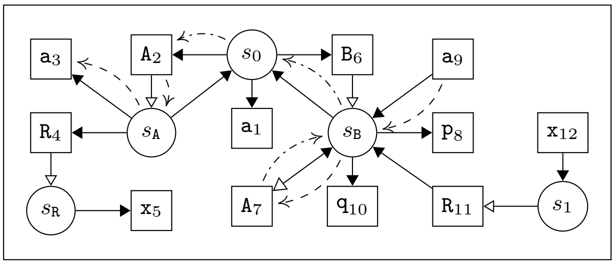
2 A Model for Name Binding: Néron et al., 2015

Scope graphs were introduced as a language-independent theory of name binding [10]. The goal is a model for the specification of name binding structure and name resolution behavior that lends itself to formal reasoning as well as practical implementation. While aiming for broad coverage of name binding features, the focus is on strong support for non-lexical binding. Name binding structure is described as a graph consisting of nodes and edges that represent scopes, references, declarations, and named imports. The theory is instantiated for specific languages by a mapping from programs to scope graphs and a resolution policy.

We use the example program and scope graph demonstrating module import in Figure 2 as a running example. Names in the program are annotated with a unique position, written as $\mathtt{x}_i$, which allow us to conveniently refer to specific *occurrences* of names. These positions are not part of the program source and do not influence resolution behavior. The example program consists of three top-level definitions, a value `a`, and two modules `A` and `B`, where the latter imports the former. The modules contain a definition of a record type `R`, and values initialized by a constant, a reference, and a projection on a record instance.

In the scope graph, scopes are depicted by circular nodes $\textcircled{$s_i$}$: s_0 represents the scope of the whole program, and $s_{\mathtt{A}}$ and $s_{\mathtt{B}}$ represent the bodies of modules `A` and `B`, respectively. The edges from $s_{\mathtt{A}}$ and $s_{\mathtt{B}}$ to s_0 indicate that s_0 is the *lexical parent* of $s_{\mathtt{A}}$ and $s_{\mathtt{B}}$, and make the declarations in s_0 available inside the module bodies. Occurrences of names are depicted as square nodes $\boxed{x_i}$. The closed arrow $\textcircled{$s_0$}\longrightarrow\boxed{\mathtt{a}_1}$ indicates a declaration (cf. $\mathtt{A}_2$ and $\mathtt{x}_5$). Reverse arrows, such as $\boxed{\mathtt{a}_9}\longrightarrow\textcircled{$s_{\mathtt{B}}$}$, indicate a reference (cf. $\mathtt{A}_7$ and $\mathtt{x}_{12}$). The open arrow $\boxed{\mathtt{A}_2}\longrightarrow\!\triangleright\textcircled{$s_{\mathtt{A}}$}$ indicates that $s_{\mathtt{A}}$ is the *associated scope* of $\mathtt{A}_2$ (cf. $\mathtt{B}_6$). The arrow $\textcircled{$s_{\mathtt{B}}$}\longrightarrow\!\triangleright\boxed{\mathtt{A}_7}$ indicates an *import* of the (still to be resolved) module `A`.

```
def a1 = 3
module A2 {
  def a3 = 21
  record R4 { x5: Int }
}
module B6 {
  import A7
  def p8 = a9
  def q10 = new R11 { x12 = 7 }.x13
}
```



Reachability $\mathbf{P}^*\mathbf{I}^*$
Visibility $\mathbf{I} < \mathbf{P}$ $\mathbf{D} < \mathbf{P}$ $\mathbf{D} < \mathbf{I}$

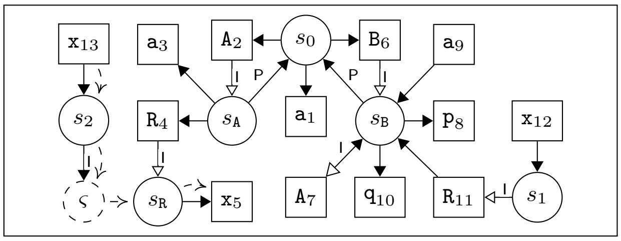
Figure 2 Example program in LMR [17] and its scope graph à la Néron et al., 2015. Dashed arrows show the resolution path of $\mathtt{a}_9$, which depends on the dash-dotted resolution path of $\mathtt{A}_7$.

References are resolved to declarations of the same name by finding paths in the graph that respect the global resolution policies. A path consists of steps: $\mathbf{P}$ when traversing a lexical parent edge, and $\mathbf{I}$ when traversing an import (after which the path continues from the associated scope of the resolved declaration). Declarations are *reachable* from a scope if the path matches the regular expression $\mathbf{P}^*\mathbf{I}^*$, which allows declarations from lexical parents, as well as declarations from (transitive) imports, but not from the lexical parents of imported scopes. Multiple reachable declarations are disambiguated to the set of *visible* declarations using the order on paths, which, in a particular scope, prefers imported declarations over parent declarations ($\mathbf{I} < \mathbf{P}$), and local declarations (represented by the terminal step $\mathbf{D}$) over either of those ($\mathbf{D} < \mathbf{P}$ and $\mathbf{D} < \mathbf{I}$). We can see the effect of the resolution policy on the resolution of reference $\mathtt{a}_9$. Two declarations are reachable from $s_\mathtt{B}$: $\mathtt{a}_1$ via a $\mathbf{P}$ step to s_0, and $\mathtt{a}_3$ via an $\mathbf{I}$ step to $s_\mathtt{A}$ (using import $\mathtt{A}_7$ which resolves to $\mathtt{A}_2$). The visibility policy prefers $\mathbf{I}$ over $\mathbf{P}$, resulting in a single resolution to declaration $\mathtt{a}_3$. This demonstrates that $\mathbf{I} < \mathbf{P}$ can be interpreted as: declarations from imports (e.g. $\mathtt{a}_3$) shadow declarations from lexically enclosing scopes.

Scopes and imports can also be used to model other binding structures, such as nominal record types. The record type declaration $\mathtt{R}_4$ is modeled similar to modules, with a scope $s_\mathtt{R}$ associated to the declaration. Record instantiation is modeled using an instance scope s_1, which imports $s_\mathtt{R}$ via import $\mathtt{R}_{11}$, which itself is resolved via import $\mathtt{A}_7$. Note that reference $\mathtt{x}_{13}$, which is the right hand side of a record *projection*, is not modeled in this scope graph. Indeed, resolving a projection requires knowledge of the *type* of the left-hand side, which was not accounted for in the original scope graph framework, but was added in the work discussed in the next section.

3 Formalizing Type Checkers based on Scope Graphs

The next step after the inception of scope graphs was incorporating them in a meta-language for type system specifications. As discussed in Section 1, this meta-language should be (i) principled; (ii) expressive; (iii) declarative; (iv) executable; (v) reuseable; and (vi) resilient [17]. These goals have governed the research discussed in this section. This meant evolving the scope graph formalism itself to increase the range of supported language features, developing declarative and operational semantics for the meta-languages, and developing reusable techniques for concurrency and incrementality that applied to all specifications written in a meta-language.



Labels
$\{\mathsf{P}, \mathsf{I}\}$

Reachability
$\mathsf{P}^*\mathsf{I}^*$

Visibility
$\mathsf{I} < \mathsf{P}$
$\$ < \mathsf{P} \quad \$ < \mathsf{I}$

$\ldots$
$\mathtt{R}_4 : \mathbf{rec}(s_{\mathtt{R}})$
$\mathtt{R}_{11} \mapsto \delta$
$\delta : \tau$
$\tau = \mathbf{rec}(\varsigma)$
$\ldots$

Figure 3 Scope graph and some of the type constraints à la van Antwerpen et al., 2016 for the example program of Figure 2. Dashed arrows show the resolution path of $\mathtt{x}_{13}$.

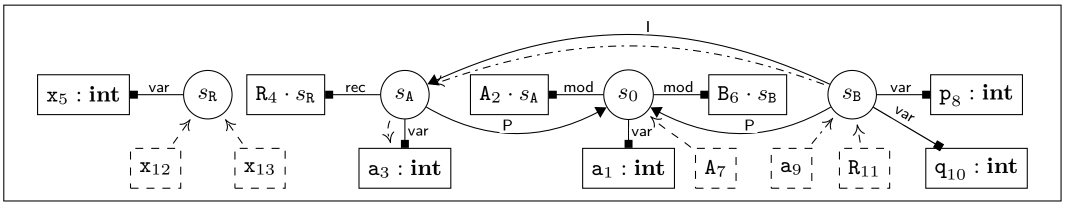
3.1 Scope Graph Constraints: van Antwerpen et al., 2016

The first step towards type checking based on scope graphs was NaBL2, a constraint language with first-class support for scope graph construction and name resolution [17]. This language consists of (i) scope graph constraints, which assert nodes and edges of the scope graph, (ii) resolution constraints, which resolve references or check properties of sets of reachable and visible declarations, and (iii) type constraints to associate types with declarations and check type equality.

From early on, it was clear that name resolution and type checking cannot always be separated into entirely distinct phases. The challenge is that type-dependent references have to be resolved in a scope that is only determined during type checking, making interleaving them inevitable. The overall approach that was chosen to support type-dependent name resolution is to allow for some variability in the structure of the scope graph: edge *targets* could be variables that are instantiated during type checking. This resulted in a two phase approach: first constraints are generated for a program, and then these constraints are solved by a solver which ensures correct sequencing of resolution and variable instantiation.

To accommodate this approach, the paper expanded the scope graph formalism in two ways. Looking at the scope graph in Figure 3, we see some notable differences compared to Figure 2. First, scope-to-scope and import edges are explicitly labeled in the graph ($\xrightarrow{\mathsf{l}}$ and $\xrightarrow{\mathsf{l}}$), generalizing the notion of fixed **P** and **I** steps. Consequently, reachability and well-formedness are expressed in terms of labels (where \$ takes the place of **D**). Custom labels are useful because they allow multiple name resolution policies to be composed. For example, using the regular expression $\mathsf{P}^*(\mathsf{TI}^*|\mathsf{I}?)$, transitive ($\mathsf{TI}$) and non-transitive ($\mathsf{I}$) imports can co-exist in the same specification. Second, the graph contains a scope *variable*, depicted as ς, that represents the context in which $\mathtt{x}_{13}$ must be resolved. As the value of ς depends on the *type* of the record instantiation, it is identified during type checking (phase 2) rather than at constraint generation time. This is demonstrated using some of the constraints of the program (shown on the right side of Figure 3). The first constraint indicates that $\mathtt{R}_4$ has type $\mathbf{rec}(s_{\mathtt{R}})$. The second constraint instantiates δ to the declaration $\mathtt{R}_{11}$ resolves to. The third constraint indicates that τ is the type of δ. Finally, the fourth constraint asserts that τ must be a record type with scope ς. During constraint solving, the solver instantiates δ with $\mathtt{R}_4$, τ with $\mathbf{rec}(s_{\mathtt{R}})$, and ς with $s_{\mathtt{R}}$, which allows the reference $\mathtt{x}_{13}$ to resolve to declaration $\mathtt{x}_5$ in $s_{\mathtt{R}}$. This illustrates how variable scopes enable type-dependent name resolution.

Despite being very expressive, this version of name resolution suffered from both theoretical and practical problems with imports. The theoretical problem, already noted by Néron et al. [10] as "anomalies", is the behavior that a single import reference might have different interpretations depending on the reference being resolved through it – not in line with the usual expectation that names have a single meaning consistent with all uses. The practical



Labels $\{\mathsf{P}, \mathsf{I}\}$ **Relations** $\{\mathsf{var}, \mathsf{mod}, \mathsf{rec}\}$

Queries	**Relation**	**Reachability**	**Visibility**	**Data**
$\mathtt{A}_7$	mod	P^*	$\$ < \mathsf{I}$	$x_i : \tau$ matches y_j if $x = y$
$\mathtt{a}_9, \mathtt{x}_{12}, \mathtt{x}_{13}$	var	$\mathsf{P}^*\mathsf{I}^*$	$\$ < \mathsf{I} < \mathsf{P}$	$x_i : \tau$ matches y_j if $x = y$
$\mathtt{R}_{11}$	rec	$\mathsf{P}^*\mathsf{I}^*$	$\$ < \mathsf{I} < \mathsf{P}$	$x_i : \tau$ matches y_j if $x = y$

Figure 4 Scope graph à la van Antwerpen et al., 2018 for the example program of Figure 2. Dash-dotted arrows show the resolution path of $\mathtt{a}_9$.

problem is that the common pattern where a reference is imported in the same scope (such as $s_{\mathtt{B}} \leftarrow\!\triangleright \boxed{\mathtt{A}_7}$) led to run times proportional to the factorial of the number of such imports in a scope. These problems were solved by the developments described in the next section.

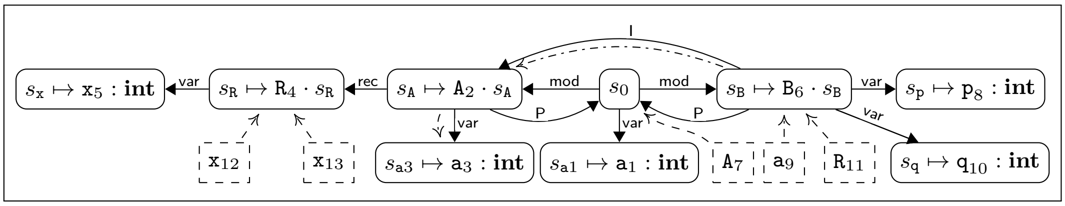
3.2 A Logic with Scope Graphs: van Antwerpen et al., 2018

While an important step forward, the previous work was mostly limited to languages with simple and nominal types. The constraint language was insufficient to model, for example, structural types and subtyping, or something like Java's nominal subtyping. This led to a radical attempt to generalize the ideas of the constraint language to a full logic. The result was a logic language, Statix, with first class support for scope graph construction and querying, which could be used to write the judgments necessary for type checking [18].

With these developments also came several substantial changes to the scope graph formalism, illustrated in Figure 4. First, the scoped relation $s_0 \xrightarrow{\mathsf{var}} \boxed{\mathtt{a}_1 : \mathbf{int}}$ generalizes declarations and type constraints. The declaration is labeled with a relation (var for variables) and can hold arbitrary data (a pair of an occurrence and its type). The other relations (mod and rec, for modules and record types, respectively), hold pairs of an occurrence and a scope reference. Second, references, which were part of the graph, are replaced by *queries*, which are ephemeral, but depicted here as $\fbox{} \dashrightarrow \bigcirc$ on top of the graph[2]. Parameters that previously were global to the formalism are now specified per query: the relation, reachability and visibility, and the predicate that matches data. The latter provides the ability to match only part of the data (ensuring we resolve a name, while ignoring its type or scope). In addition, having a single relation for each query also simplifies namespacing. These generalizations led to a great increase in expressivity, evidenced by case studies for structural records, nominal subtyping and parametric polymorphism.

One significant change to the scope graph formalism is the elimination of explicit imports. It was noticed that one could write predicate rules that mimic imports by querying the import reference, and asserting a direct edge to the resulting associated scope. The semantics of Statix then ensure that every reference will get a single interpretation, thereby side-stepping both the anomalies and the performance issues of NaBL2. Yet, as a consequence, the graph resembles the original program less: imports become simple edges between scopes, and their dependency on import references is no longer visible.

[2] Contrary to [18], we use dashed lines to emphasize that queries are not part of the graph.


$s_x \mapsto x_5$: int
var
$s_R \mapsto R_4 \cdot s_R$
rec
$s_A \mapsto A_2 \cdot s_A$
mod
s_0
mod
$s_B \mapsto B_6 \cdot s_B$
var
$s_p \mapsto p_8$: int
I
P
P
var
var
var
x_{12}
x_{13}
$s_{a3} \mapsto a_3$: int
$s_{a1} \mapsto a_1$: int
A_7
a_9
R_{11}
$s_q \mapsto q_{10}$: int

Labels {P, I, var, mod, rec}

Queries	**Reachability**	**Visibility**	**Data**
A_7	P^*mod	mod < P	$x_i : \tau$ matches y_j if $x = y$
a_9, x_{12}, x_{13}	P^*I^*var	var < I < P	$x_i : \tau$ matches y_j if $x = y$
R_{11}	P^*I^*rec	rec < I < P	$x_i : \tau$ matches y_j if $x = y$

Figure 5 Scope graph à la Rouvoet et al., 2020 for the example program of Figure 2. Dash-dotted arrows show the resolution path of a_9.

3.3 Sound Scheduling: Rouvoet et al., 2020

A major challenge resulting from the development of Statix was how to ensure sound execution of Statix programs. Recall that one goal was to develop meta-languages that were both formal and declarative, as well as executable. The fact that Statix allows edge construction to depend on query resolution is a challenge for the latter. Compared to NaBL2, where only edge *targets* might be unknown, Statix execution might need to resolve queries when large parts of the graph are still missing. The idea of *critical edges*, which are edges that are part of the final graph, but not yet present in the current, partial graph, provides a strategy for execution and allows reasoning about its soundness. These edges are sufficient to determine if a query result in the partial graph is sound (i.e., equal to its result in the final graph). This idea is developed into an execution strategy for Statix that over-approximates critical edges using a static analysis of a specification to delay query resolution until soundness is guaranteed. Due to the over-approximation, the algorithm is incomplete, and it can get stuck on typeable programs. While generally not a problem in practice, there have been a few cases where some parts of the specification needed to be rewritten to work around this.

To aid reasoning, Rouvoet et al. present a leaner formalization of both scope graphs and Statix, called Statix-core. We illustrate this with the example scope graph in Figure 5. Instead of scopes and declarations, there is just a single node type, which combines identity and an optional data term, depicted as $\boxed{s_i \mapsto d}$. Edge and relations labels are combined, and the relation parameter of the query becomes part of the reachability regular expression. The query algorithm is simplified by disambiguating after, instead of during, resolution.

3.4 Concurrency: van Antwerpen et al., 2021

Although Statix is an expressive specification language, its type checkers turn out to be rather slow. The first attempt to mitigate this problem was to execute type checkers *concurrently* [19]. A concurrent execution model based on scope graphs is developed, that can serve as the basis for Statix execution. The scope graph formalism is extended with a notion of first-class compilation units, where each unit "owns" part of the scope graph. Units are organized hierarchically, with *shared scopes* connecting parent and child units' scope graphs. Usually, the root unit corresponds to the whole project, the leaves to files, and the units in between to packages or modules. Compilation units are mapped to actors, the units of parallel computation, each of which runs its own type checker. The type checkers use an API to construct and query their scope graph, while resolution queries in other units are implemented using message passing. Query scheduling is coordinated using *scope states*, a generalization of the critical edges of Rouvoet et al. [16]. Type checkers initialize fresh scopes with a set of open (critical) edges, which monotonically decreases as the type checkers closes edges (i.e., marks them non-critical). By translating changes in the set of critical edges to scope state operations, the Statix solver was ported to use the concurrent model easily. The concurrent execution can also suffer from incompleteness if open edges are over-approximated. Termination is ensured using a deadlock detection and resolution scheme.

3.5 Incrementality: Zwaan et al., 2022

This concurrent framework was extended to have compilation unit level *incrementality* as well [28]. When type checking a project incrementally, *edited* units compute a *diff* of their scope graphs (i.e., a set of added and removed scopes and edges), which is used to compute differences in query answers (i.e., added or removed paths) of *unchanged* units. Such units require reanalysis only when an answer to an outgoing query changed. By extending the deadlock resolution of the concurrent solver, the algorithm can deal with mutually recursive dependencies correctly. Case studies resulted in speedups up to 150x on synthetic Java projects and up to 20x on real-world commits from Java and WebDSL projects.

3.6 Partial Evaluation: Zwaan, 2022

Statix describes name resolution using some high-level declarative parameters, such as reachability regular expressions and label orders. This allows expressing reachability and visibility policies concisely, but induces significant overhead when resolving queries. As these parameters are known statically (i.e., no query *parameters* are computed dynamically), partial evaluation [4] can be applied to the query resolution algorithm [27]. This yields imperative query resolution functions, that implement a query with fixed parameters. As such, these specialized queries do not induce overhead from interpreting query parameters anymore. In case studies on Java projects, the type checker runtime decreased 38% to 48%.

4 Beyond Type Checking

In this section, we discuss how declarative specifications based on scope graphs have been used for the dynamic semantics of a language, editor services, and refactorings.

Dynamic Semantics. The layout of runtime heaps often corresponds to the static binding structure of a program. This notion is made precise in the "scopes describe frames" paradigm [14], in which a heap is defined as a collection of frames. Each frame corresponds to a scope, with links to other frames that correspond to edges in a scope graph, and slots with values that correspond to declarations. This establishes a systematic, language-parametric relation between runtime memory layout and static binding. The correspondence between scopes and frames is used to prove language properties such as type soundness, as well as providing a framework for safe garbage collectors.

The "scopes describe frames" paradigm has been used to define intrinsically-typed definitional interpreters for imperative languages [15]. While type-safety proofs for languages with mutable state are usually challenging, using this framework, most of the proof work could be delegated to the (dependently-typed) host language. Finally, Vergu et al. [21] show that mapping scopes and frames to the Truffle Object Storage Model allows a significant speedup of meta-interpreters.

Editor Services. Interactive editor services are as important as a compiler to a modern development experience. Declarative meta-languages, which can be understood independent of their implementations, allow us to derive such editor services as alternative interpretations of a specification [12]. For example, Statix is used for a language-parametric, sound and complete approach to code completion [11]. Type-sound proposals are generated by turning the proposal position into a unification variable, and applying search strategies to the constraint problem, to find possible instantiations. These instantiations are then translated back to completion proposals. The implementation is language-parametric, and can be reused as an off-the-shelf component for actual language implementations.

Applying scope graphs as the backbone of language-parametric implementations of common refactorings has been explored as well. This resulted in approaches for *renaming* [9] and function *inlining* [20]. While this demonstrates that scope graphs can facilitate refactorings, they have shown a limitation of Statix' queries. In particular, one cannot express a query that resolves all names that would result in capture. Thus, these refactorings perform a full reanalysis of the program during or after the transformation, which yields significant performance overhead.

Cross-Language Type Checking. Since scope graphs provide a uniform representation of name binding, exploratory research in cross-language type checking within a project has been conducted [26]. This study suggests that scope graphs generated by specifications of different languages are composable when the specifications agree on an "interface", which is a shared collection of labels, declarations and resolution policies. However, approximating critical edges [16] and determining rule selection order in Statix require whole-program analyses. Therefore, Statix' specifications turn out to be composable only when the rule sets of the fragments are disjoined, limiting the expressiveness of the approach. Perhaps this approach can be adapted to use compilation units, which executes each unit with its own type checker, making specification composition unnecessary.

5 Evaluation

The work discussed in the previous sections shows the historical development from imperative approaches to name binding using Stratego to a family of meta-languages and language-parametric sevices based on the scope graph formalism. But to what extend has the goal of declarative specification of rich name binding patterns been achieved? In this section, we evaluate the formalism using the goals from Section 1.

Principled. We first consider whether the developed theory is "clear and clean". As these criteria are a little subjective, and no user evaluation has been conducted at the moment of writing, we mainly base this evaluation on informal feedback acquired over the years. First, the scope graph theory is small, has a well-defined semantics, and closely relates to well-known notions from graph theory. For good scope graph models, the elements of a scope graph (nodes, declarations and edges) can intuitively be related to the original program. Its main weakness is its formulation in terms of individual references, which leads to the anomalous behavior that allows multiple interpretations of import references depending on the the reference resolved through the import. Second, the meta-languages are relatively small, have well-defined semantics, and stay close to existing concepts of constraint and logic programming. Specifications can be understood despite the complexity that is necessary to execute them.

Expressive. We begin to consider the expressiveness of scope graphs as a model of name binding. Case studies show that many different name binding patterns can be encoded, ranging from sequential, parallel and recursive let-bindings to transitive and non-transitive module imports using full or partial qualifiers [10] as well as nominal and structural records with extension and subtyping [17, 18]. Some important limitations we observed so far: First, substructural type systems are hard to encode, because scope graphs cannot express constraints on declaration access count and ordering. Second, more complex shadowing policies, such as Scala's preference of outer named imports over more closely nested wildcard imports, cannot be expressed with simple label orders. Third, the work on renaming and inlining suggests that scope graph queries are not expressive enough to accommodate concise implementations of refactorings, although precise requirements are still unclear.

Next we consider the expressiveness of the meta-languages, in particular Statix. Case studies show that a variety of typing relations can be expressed, ranging from type-dependent names [17], to parametric polymorphism [18], as well as disambiguation of qualifiers [19]. In addition, major limitations exist around inference. First, the absence of support for reasoning about or abstracting over free unification variables makes it impossible to specify Hindley-Milner-style type inference. Second, it is not possible to infer bits of scope graph, not even for fairly local cases such as a record type based on its usage in a function body.

Declarative. The meta-languages abstract over implementation concerns such as staging and scheduling constraint resolution. Their specifications can be understood in terms of declarative semantics [16, fig. 7], which are free of the complexities required for the operational semantics. However, as we will see in the next section, there are rare cases where the specification was changed to accommodate the incompleteness of the solver algorithm. Although the declarative semantics were intended to support formal reasoning, we have only seen that for soundness of the meta-languages themselves, and for definitional interpreters that use customized scope graph representations [14, 15]. Reasoning about properties of individual language specifications, other interpreters, or proving properties of individual programs based on language semantics, is still mostly unexplored.

We also consider whether specifications have a desirable level of abstraction. For many of the motivating use cases that we already mentioned, Statix allows clean and concise encodings, and the rules are quite close to traditional pen-and-paper inference rules. For some cases, the encodings can be very verbose, and the intent gets lost. Examples are parametric polymorphism (e.g., Featherweight Generic Java), which requires explicit substitution logic, and the disambiguation of syntactically ambiguous references in Java, which requires verbose decision procedures that lack the conciseness of the scope graph shadowing primitives.

Executable. The need for practical algorithms has always been the sword of Damocles, hanging over us when we consider new theories. While all the research presented before has always addressed both theory and practice, each has sometimes suffered for the others sake. Regarding scope graphs we consider the following points. First, the original resolution algorithm [10, 17] could perform very poorly when multiple imports were present in a single scope. Circumventing this problem by dropping imports from scope graphs solved this, but has weakened scope graphs as a stand-alone model for name binding: name-based dependencies are not clearly reflected in the graph anymore, and it depends more on the embedding meta-language for common patterns. Second, the scope graph theory has always assumed path orders to be lexicographical orders based on edge labels, and the resolution algorithms handle shadowing locally decided based on outgoing edge labels. The Scala case study [16] shows that local shadowing is not sufficient. Allowing full path ordering would be an innocuous change to the theory, but the performance impact on the resolution algorithms is unclear. Third, the resolution algorithm operates on single references or queries. This inhibits caching, resulting in poor performance, as many parts of the graph are traversed multiple times. This is a problem in practice, since a significant part of meta-language runtime is determined by scope graph resolution. It is an open question whether a resolution algorithm is possible that supports explicit imports without suffering from "anomalies", and what the impact on the theory would be.

Both NaBL2 and Statix allow deriving executable type checkers, which are provably correct (i.e., return results sound with respect to the declarative semantics). However, the need to interleave graph construction and graph querying in Statix because of the absence of imports, has greatly complicated its operational semantic. Additionally, the over-approximation of critical edges can lead to a stuck solver on type-correct programs. Most notably, this was observed when studying the module system of Rust, which was expressible in Statix, but could not be made executable. Regarding performance, Statix has been designed to avoid excessive run times, for example by eliminating backtracking, but so far Statix-based type checkers are still an order of magnitude slower than hand-written ones.

Reusable. Scope graphs and Statix abstract over common type checker implementation concerns, such as implementing name binding operations, first-order unification, as well as staging and scheduling, which are reusable through their implementations. In addition, the formalism enabled reuse of editor services such as code completion and refactorings. Still, specifications sometimes have significant boilerplate, due to the lack of sharing possibilities in the meta-language itself. This is a limitation of the current implementation rather than the approach. In the future, we envision supporting polymorphic predicates in the Statix surface language to facilitate reuse of constraints. In addition, a notion of "specification libraries", where we can standardize and reuse particular type system features, could encourage reuse, not just of code, but also language concepts, beyond individual specifications.

Resilient. Scope graph resolution is resilient to erroneous and incomplete programs, and will simply return partial or ambiguous results. The meta-languages, in a similar manner, try to solve as much of the type checking problem as possible, while collecting error messages for failed constraints. This is useful when iteratively developing a language. However, the behavior on erroneous programs is not formalized. Thus, there are no guarantees on the behavior of these type checkers on incorrect programs. Similarly, the clarity of the reported errors varies a lot. Especially the combination of unification and dynamic constraint scheduling can result in unexpected errors, that are hard to debug. Improving the quality of error messages is therefore an important open research question.

6 Conclusion

In this paper, we have provided an overview of Eelco Visser's research line related to scope graphs and identified many of its strengths and opportunities for future improvement. While many interesting questions remain, scope graphs have shown to be a solid and reliable foundation for both understanding and implementing name binding related components of language implementations.

References

1 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.

2 Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2006. URL: https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06.

3 Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming type rules. In *8th International Symposium on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2006. doi:10.1007/11737414_4.

4 Yoshihiko Futamura. Partial computation of programs. In *RIMS Symposium on Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 1982. doi:10.1007/3-540-11980-9_13.

5 Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

6 Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402, 2010. doi:10.1007/s10270-009-0136-1.

7 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869497.

8 Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *5th International Conference on Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. doi:10.1007/978-3-642-36089-3_18.

9 Phil Misteli. Renaming for everyone: Language-parametric renaming in spoofax. Master's thesis, Delft University of Technology, May 2021. URL: http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5.

10 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems – 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

11 Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA):1–30, 2022. doi:10.1145/3527329.

12 Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. Towards language-parametric semantic editor services based on declarative type system specifications (brave new idea paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming*, volume 134 of *LIPIcs*. Dagstuhl, 2019. doi:10.4230/LIPIcs.ECOOP.2019.26.

13 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

14 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.20.

15 Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158104.

16 Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428248.

17 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

18 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.

19 Hendrik van Antwerpen and Eelco Visser. Scope states: Guarding safety of name resolution in parallel type checkers. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.1.

20 Loek van der Gugten. Function inlining as a language parametric refactoring. Master's thesis, Delft University of Technology, June 2022. URL: http://resolver.tudelft.nl/uuid:15057a42-f049-4321-b9ee-f62e7f1fda9f.

21 Vlad A. Vergu, Andrew P. Tolmach, and Eelco Visser. Scopes and frames improve meta-interpreter specialization. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.4.

22 Eelco Visser. A theory of name resolution (blog), January 2015. URL: https://web.archive.org/web/20220925104204/https://eelcovisser.org/blog/writing/2015/01/30/a-theory-of-name-resolution/ [cited 18-01-2023].

23 Eelco Visser. Understanding software through linguistic abstraction. *Science of Computer Programming*, 97:11–16, 2015. doi:10.1016/j.scico.2013.12.001.

24 Eelco Visser. Fast and safe linguistic abstraction for the masses. In Marieke Huisman, Wouter Swierstra, and Eelco Visser, editors, *Tech Report UU-CS-2019-004: A Research Agenda for Formal Methods in the Netherlands*, pages 10–11. Department of Information and Computing Sciences, Utrecht University, 2019.

25 Guido Wachsmuth, Gabriël Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering – 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013. doi:10.1007/978-3-319-02654-1_15.

26 Aron Zwaan. Composable type system specification using heterogeneous scope graphs. Master's thesis, Delft University of Technology, January 2021. URL: http://resolver.tudelft.nl/uuid:68b7291c-0f81-4a70-89bb-37624f8615bd.

27 Aron Zwaan. Specializing scope graph resolution queries. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3567512.3567523.

28 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2), 2022. doi:10.1145/3563303.