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.

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.