Report from Dagstuhl Seminar 16131

Language Based Verification Tools for Functional Programs

Edited by

Marco Gaboardi[1], Suresh Jagannathan[2], Ranjit Jhala[3], and Stephanie Weirich[4]

1 **SUNY – Buffalo, US,** `gaboardi@buffalo.edu`
2 **Purdue University – West Lafayette, US,** `suresh@cs.purdue.edu`
3 **University of California – San Diego, US,** `jhala@cs.ucsd.edu`
4 **University of Pennsylvania – Philadelphia, US,** `sweirich@cis.upenn.edu`

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 16131 "Language Based Verification Tools for Functional Programs". This seminar is motivated by two converging trends in computing – the increasing reliance on software has led to an increased interest in seeking formal, reliable means of ensuring that programs possess crucial correctness properties, and the dramatic increase in adoption of higher-order functional languages due to the web, multicore and "big data" revolutions.

While the research community has studied the problem of language based verification for imperative and first-order programs for decades – yielding important ideas like Floyd-Hoare Logics, Abstract Interpretation, Model Checking, and Separation Logic and so on – it is only relatively recently, that proposals have emerged for language based verification tools for functional and higher-order programs. These techniques include advanced type systems, contract systems, model checking and program analyses specially tailored to exploit the structure of functional languages. These proposals are from groups based in diverse research communities, attacking the problem from different angles, yielding techniques with complementary strengths.

This seminar brought diverse set of researchers together so that we could: compare the strengths and limitations of different approaches, discuss ways to unify the complementary advantages of different techniques, both conceptually and in tools, share challenging open problems and application areas where verification may be most effective, identify novel ways of using verification techniques for other software engineering tasks such as code search or synthesis, and improve the pedagogy and hence adoption of such techniques.

Seminar March 28 to April 1, 2016 – http://www.dagstuhl.de/16131
1998 ACM Subject Classification D.2.4 [Software Engineering]: Software/Program Verification F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs
Keywords and phrases Functional Programming, Type Systems, Contracts, Dependent Types, Model Checking, Program Analysis
Digital Object Identifier 10.4230/DagRep.6.3.59

1 Summary

Marco Gaboardi
Suresh Jagannathan
Ranjit Jhala
Stephanie Weirich

This report summarizes the program and the outcomes of Dagstuhl Seminar 16131 "Language Based Verification Tools for Functional Programs", organized by:
- Marco Gaboardi, School of Computing, University of Dundee, UK
- Suresh Jagannathan, Purdue University, USA
- Ranjit Jhala, University of California, San Diego, USA
- Stephanie Weirich, University of Pennsylvania, USA.

The web, multi-core and "big-data" revolutions have been largely built on higher-order programming constructs pioneered in the Functional Programming community. Despite the increasing importance of such programs, there are relatively few tools that are focussed on ensuring that functional programs possess crucial correctness properties. While language based verification for imperative and first-order programs has been studied for decades yielding important ideas like Floyd-Hoare Logics, Abstract Interpretation, and Model Checking. It is only relatively recently, that researchers have proposed language based verification tools e.g. advanced type systems, contract systems, model checking and higher-order program analyses for functional and higher-order programs.

We organised this seminar to bring together the different schools of researchers interested in software reliability, namely, the designers and implementers of functional programming languages, and experts in software verification, in order create a larger community of researchers focused on this important goal, to let us compare the strengths and limitations of different approaches, to find ways to unite both intellectually, and via tools the complementary advantages of different techniques, and to devise challenging open problems and application areas where verification may be most effective. To this end, the seminar comprised a program of 30 talks from the leading experts on the above topics, and breakout sessions on:
1. Integrating formal methods tools in the curriculum
2. Hands on Tool Tutorials
3. User Interaction
4. Types and Effects

2 Table of Contents

3 Overview of Talks

3.1 Coinduction using copatterns and sized types

Andreas Martin Abel (Chalmers UT – Göteborg, SE)

Joint work of Andreas Martin Abel, Brigitte Pientka, Anton Setzer, and David Thibodeau
Main reference A. Abel, B. Pientka, "Wellfounded recursion with copatterns: a unified approach to termination and productivity", in Proc. of the 18th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'13), pp. 185–196, ACM, 2013.
URL http://dx.doi.org/10.1145/2500365.2500591

I present the new coinduction mechanism of Agda based on copatterns and sized types (Abel et al., POPL 2013, Abel/Pientka, ICFP 2013). As a running example, I use a coinductive definition of formal languages. I show how to use infinite tries to represent languages (sets of strings) and demonstrate that Agda allows an elegant representation of the usual language constructions like union, intersection, concatenation, and Kleene star. I also show how to reason about equality of languages using bisimulation and coinductive proofs.

3.2 Verified Compilers for a Multi-Language World

Amal Ahmed (Northeastern University – Boston, US)

Joint work of Amal Ahmed; James Perconti
Main reference A. Ahmed, "Verified Compilers for a Multi-Language World", in Proc. of the 1st Summit on Advances in Programming Languages (SNAPL 2015), LIPIcs, Vol. 32, pp. 15–31, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
URL http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.15

Verified compilers are typically proved correct under severe restrictions on what the compiler's output may be linked with, from no linking at all to linking only with code compiled from the same source language. Such assumptions contradict the reality of how we use these compilers since most software systems today are comprised of components written in different languages compiled by different compilers to a common target, as well as low-level libraries that may be handwritten in the target language.

The key challenge in verifying compilers for today's world of multi-language software is how to formally state a compiler correctness theorem that is compositional along two dimensions. First, the theorem must guarantee correct compilation of components while allowing compiled code to be composed (linked) with target-language components of arbitrary provenance, including those compiled from other languages (horizontal compositionality). Second, the theorem must support verification of multi-pass compilers by composing correctness proofs for individual passes (vertical compositionality).

In this talk, I'll describe a new methodology for building compositional verified compilers for today's world of multi-language software and discuss the challenges that lie ahead. Our project has two central themes, both of which stem from a view of compiler correctness as a language interoperability problem. First, to specify correctness of component compilation, we require that if a source component S compiles to target component T, then T linked with some arbitrary target code T' should behave the same as S interoperating with T'. The latter demands a formal semantics of interoperability between the source and target languages. Second, to enable safe interoperability between components compiled from languages as different as ML, Rust, C, and Coq's Gallina, we plan to design a gradually type-safe target language based on LLVM that supports safe interoperability between more precisely typed, less precisely typed, and type-unsafe components.

3.3 DeepSpec, CertiCoq and Verified Functional Algorithms

Andrew Appel (Princeton University – US)

I will speak briefly on three different topics: The new NSF-funded project at Princeton/Penn/Yale/MIT "The Science of Deep Specification," the CertiCoq project (Appel/Morrisett/-Pollack/Sozeau and students) to formalize the extraction and compilation process, and my new interactive-in-Coq textbook in the Software Foundations series, "Verified Functional Algorithms."

3.4 Type Systems as Proof Strategies

Iavor Diatchki (Galois Systems – Portland, US)

I'd like to share some thoughts on the design of type systems for existing dynamically typed languages such as Lua, JavaScript, Python, etc. The core idea is to blur the line between formal-verification and type-checking, and try to present a type system as a library of strategies that are able to discharge certain proof obligations. This is part of my ongoing research, and the ideas are not yet fully realized, but – if possible – I'd like to share them to get feedback and advice by fellow researchers.

3.5 Dependently Typed Programming in GHC 8

Richard A. Eisenberg (University of Pennsylvania – Philadelphia, US)

This talk demonstrates some of the new features I have added in the newest release of the Glasgow Haskell Compiler (GHC 8). GHC 8 supports the (Type : Type) axiom and visible type application, allowing easier and more expressive dependently typed programs than were possible previously. The main example in this talk is a program that performs type-safe database access while inferring the desired database schema from the program text. By inferring the schema, this program contains less boilerplate code and is more flexible than other type-safe database access approaches would allow.

3.6 Deductive Verification with Why3

Jean-Christophe Filliâtre (CNRS & University Paris Sud, FR)

This short talk is a brief overview of Why3, a tool for deductive program verification.

Why3 provides an imperative programming language (with polymorphism, algebraic data types, pattern matching, exceptions, references, arrays, etc.), a mathematical language that is an extension of first-order logic, and a technology to discharge verification conditions using several, off-the-shelf interactive and automated theorem provers (Coq, Alt-Ergo, Z3, CVC3, etc.).

More details at http://why3.lri.fr/.

3.7 Relational cost analysis

Deepak Garg (MPI-SWS – Saarbrücken, DE)

Many applications require analysis of the relative use of resources (time, space) by two different programs or the same program with two different inputs. We call this the problem of relational cost analysis. Examples include compiler optimizations, incremental computation, static detection of side-channel leaks in security-critical programs and the stability analysis of algorithm implementations. This talk presents the beginnings of a type theory for relational cost analysis. It explains how a combination of lightweight dependent types, ideas from information flow analysis and a bit of co-monadic analysis can be combined to perform such analysis on non-trivial examples.

3.8 The Interactive Software Verification System KeY

Reiner Hähnle (TU Darmstadt, DE)

A brief overview of how KeY works, what can be done with it, the degree of automation, the user interfaces

3.9 Dependent Types and Multi-Monadic Effects in F*

Cătălin Hriţcu (INRIA – Paris, FR)

Joint work of Karthikeyan Bhargavan; Antoine Delignat-Lavaud; Simon Forest; Cédric Fournet; Cătălin Hriţcu; Chantal Keller; Markulf Kohlweiss; Aseem Rastogi; Pierre-Yves Strub; Nikhil Swamy; Santiago Zanella-Béguelin; Jean-Karim Zinzindohoue
Main reference N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J. K. Zinzindohoue, S. Zanella Béguelin, "Dependent types and multi-monadic effects in F", in Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'16), pp. 256–270, ACM, 2016; pre-print and supplementary material available from author's webpage.
URL http://dx.doi.org/10.1145/2837614.2837655
URL https://www.fstar-lang.org/papers/mumon/

We present a new, completely redesigned, version of F*, a language that works both as a proof assistant as well as a general-purpose, verification-oriented, effectful programming language.

In support of these complementary roles, F* is a dependently typed, higher-order, call-by-value language with *primitive* effects including state, exceptions, divergence and IO. Although primitive, programmers choose the granularity at which to specify effects by equipping each effect with a *monadic*, predicate transformer semantics. F* uses this to efficiently compute weakest preconditions and discharges the resulting proof obligations using a combination of SMT solving and manual proofs. Isolated from the effects, the core of F* is a language of pure functions used to write specifications and proof terms – its consistency is maintained by a semantic termination check based on a well-founded order.

We evaluate our design on more than 55,000 lines of F* we have authored in the last year, focusing on three main case studies. Showcasing its use as a general-purpose programming language, F* is programmed (but not verified) in F*, and bootstraps in both OCaml and F#. Our experience confirms F*'s pay-as-you-go cost model: writing idiomatic ML-like code with no finer specifications imposes no user burden. As a verification-oriented language, our most significant evaluation of F* is in verifying several key modules in an implementation of the TLS-1.2 protocol standard. For the modules we considered, we are able to prove more properties, with fewer annotations using F* than in a prior verified implementation of TLS-1.2. Finally, as a proof assistant, we discuss our use of F* in mechanizing the metatheory of a range of lambda calculi, starting from the simply typed lambda calculus to System F$_\omega$ and even a sizeable fragment of F* itself – these proofs make essential use of F*'s flexible combination of SMT automation and constructive proofs, enabling a tactic-free style of programming and proving at a relatively large scale.

Talk slides available at:
http://materials.dagstuhl.de/files/16/16131/16131.CatalinHritcu.Slides.html

3.10 Relational Reasoning about Higher-Order Shape Properties

Gowtham Kaki (Purdue University – West Lafayette, US) and Suresh Jagannathan (Purdue University – West Lafayette, US)

Main reference G. Kaki, S. Jagannathan, "A Relational Framework for Higher-order Shape Analysis", in Proc. of the 19th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'14), pp. 311–324, ACM, 2014.
URL http://dx.doi.org/10.1145/2628136.2628159

In this talk, I present CATALYST, a relational reasoning framework integrated within a dependent type system that is capable of automatically verifying complex invariants over the shapes of algebraic datatypes. A novel contribution of CATALYST is the concept of parametric relations – relations parameterized over other relations that enable intuitive specifications for higher-order polymorphic functions, while retaining the compositionality and decidability of type checking. I describe an encoding of fully instantiated parametric relations in a decidable subset of first-order logic, and show how it is sufficient to type check higher-order functions against expressive specifications. I describe an implementation of CATALYST in SML, present multiple examples, and end the talk with a brief note on inferring rich dependent types in CATALYST.

3.11 Program Verification Based on Higher-Order Model Checking

Naoki Kobayashi (University of Tokyo – Tokyo, Japan)

Higher-order model checking has recently been applied to fully automated verification of higher-order functional programs. In the talk, I plan to provide a tutorial to explain what is higher-order model checking and how it can be applied to program verification. I will also summarize recent progress on the higher-order model checking approach to program verification.

3.12 Certified Automated Theorem Proving for Type Inference

Ekaterina Komendantskaya (Heriot-Watt University – UK)

Joint work of Peng Fu; Ekaterina Komendantskaya; Tom Schrijvers
URL http://staff.computing.dundee.ac.uk/katya/CoALP/

Two facts are universally acknowledged: critical software must be subject to formal verification and modern verification tools need to scale and become more user-friendly in order to make more impact in industry.

There are two major styles of verification: (*) algorithmic: verification problems are specified in an automated prover, e.g. (constraint) logic programming or SMT solver, and properties of interest are verified by the prover automatically. Such provers can be fast, but their trustworthiness is hard to establish without producing and checking proofs. This is due to complexity of modern-day solvers, e.g. SMT solvers have codebases 100K in C++. These tools exhibit bugs and are not trustworthy enough for critical systems.

An alternative is (**) a typeful approach to verification: instead of verifying programs in an external prover, a programmer may record all properties of interest as types of functions in his programs. Thanks to Curry-Howard isomorphism, type inhabitants also play the role of executable functions and proof witnesses, thus completing the certification cycle.

At their strongest, types can cover most of the properties of interest to the verification community, e.g. recent dialects Liquid Haskell and F* allow pre- and post-condition formulation at type level. But, when properties expressed at type level become rich, type inference engines have to assume the role of automated provers, e.g. Liquid Haskell and F* connect directly to SMT solvers. Thus, once again, we delegate trust without having proper certification of automated proofs.

This talk was about our recent work [Fu, Komendantskaya, Schrijvers, in FLOPS 2016] that resolves the above dichotomy "scale versus trust" by offering a new, typeful, approach to automated proving for type inference. Recently, we designed a new method of using logic programming in Haskell type class inference: Horn clauses can be represented as types, and proofs by resolution – as proof terms inhabiting the types. Thus, the problem of automated inference in Horn Clause logic is re-cast as the problem of type inhabitation in a suitable type system. In this way, outputs of the resulting Curry-Howard Horn Clause prover are directly compatible with type system of Haskell's compiler. Overall, this method allows to achieve both high standards of automated proof certification and compatibility of the automated prover with the target compiler.

The question is: can this method apply to other existing algorithmic and typeful approaches?

3.13 Ramsey-based Methods: From Size-Change Termination to Satisfiability in Temporal Logics

Martin Lange (Universität Kassel, DE)

Joint work of Oliver Friedmann; Felix Klaedtke; Martin Lange
Main reference O. Friedmann, F. Klaedtke, M. Lange, "Ramsey-Based Inclusion Checking for Visibly Pushdown Automata", ACM Transactions on Computational Logic (TOCL), Vol. 16(4), Article No. 34, 2015.
URL http://dx.doi.org/10.1145/2774221

Ramsey's Theorem about the existence of infinite monochromatic subgraphs in the finitely coloured graph on the natural numbers was a crucial ingredient in Büchi's original proof of the complementation closure of the class of omega-regular languages. For most of the time since then, this has just been seen as a mathematical tool for obtaining a proof.

In the early 2000s, Lee, Jones and Ben-Amram introduced the size-change termination principle – a method for proving termination of abstract functional programs that can only reduce, copy and permute their arguments. They showed that the problem could be solved by a reduction to the inclusion problem for Büchi automata, yet the existing algorithms based on explicit complementation were not competitive enough in practice. They devised a method whose correctness directly relies Ramsey's Theorem which essentially showed that Büchi's original complementation proof has more to offer than just mathematical truth, but that it can also lead to elegant and practical algorithms for the analysis of omega-automata. Since then, Ramsey-based methods have proved to be very efficient for such tasks.

Lately, Friedmann, Klaedtke and myself have shown that Ramsey-based methods can be used effectively and efficiently beyond the class of omega-regular languages, namely for visibly pushdown omega-languages recognised by corresponding parity automata.

It still remains to be seen whether a similar development is possible for tree automata, namely whether there are successful program analysis techniques which could similarly lead to better algorithms for tree automata inclusion.

3.14 Automated verification of functional programs

Rustan Leino (Microsoft Research – USA)

Dafny started as an imperative language with specifications, where the specifications had mathematical elements also found in functional languages. These functional features grew beyond uses in specifications and now include datatypes, co-datatypes, recursive functions, and predicates defined as least/greatest fixpoints. This means Dafny can be used as a functional language. Dafny also includes an automated verifier, and the language has proof-authoring support for when automation doesn't hold up. In difference to some other functional languages with verification support, the logic of Dafny is not based around dependent types but rather Hoare logic. In this talk, I'll demo a tour through Dafny's features and point out some limitations.

3.15 A Static Type Analysis for Lua

Jan Midtgaard (Technical University of Denmark – Lyngby, DK)

Higher-order, dynamically-typed programming languages are flexible but come at the price of less tool support. To address this challenge we develop a static type analysis for the Lua programming language. Lua represents an interesting, yet minimal mix of imperative, functional, and object-oriented language features which makes this a challenging task. We present a prototype implementation of the developed analysis.

3.16 Subtle points

Conor McBride (University of Strathclyde – UK)

I'll sketch the approach to computation in Homotopy Type Theory that we're currently exploring at Strathclyde. In the business of constructing paths between isomorphic types, we introduce a notion of segmentation: nontrivially segmented paths have intermediate points and can thus be constructed in a piecewise continuous manner. Segmentations are naturally ordered by their subtlety, as any segment can be split in two. Crucially, segmented paths deliver the means to transport values between any two of their points.

3.17 Higher-order horn clauses and higher-order model checking

Luke Ong (University of Oxford – Oxford, UK)

Joint work of Luke Ong; Steven Ramsay

Higher-order model checking (HOMC) is the model checking of trees generated by recursion schemes. In the (standard) intersection type approach to HOMC, one would construct a certain type environment, which constitutes a symbolic representation of the invariant. In this ongoing work, we consider the problem of finding higher-order inductive invariants in a purely logical setting, namely, the satisfiability problem for (constrained) higher-order horn clauses. Formulated as higher-order constraint solving, the problem has a much broader appeal than recursion scheme model checking, yet we argue that much of the technology already developed by the HOMC community can be made highly effective at solving it. In particular, we describe an adaptation of Kobayashi's Hybrid Algorithm to the problem and highlight its similarities to McMillan's Lazy Annotation algorithm (for solving first-order constrained horn clauses).

3.18 From analysis-directed semantics to specifications-in-types

Dominic Orchard (University of Cambridge, UK)

Various recent works on effects and resource usage (coeffects) have provided semantic models that are indexed in some way by analysis information, e.g., by effect systems, Hoare logic triples, resource bounds. Such models typically provide inductive families of denotations, following the shape of an inductively defined program analysis. For example, "graded monads" are indexed by a monoidal effect system. This is a powerful new paradigm as it provides a way to refine semantics by analysis information, and exposes analysis information in semantics. In this talk, I give an overview of the general approach, which I call analysis-directed semantics. I then show how such models can be used directly in programming, where the indices of these semantic structures can be used to embed functional specifications at the type-level. I'll give some examples in Haskell including effects, computational complexity, communication safety for concurrency, and well-bracketed file handlers.

3.19 Programming Coinductive Proofs Using Observations

Brigitte Pientka (McGill Universitiy – Montreal, Canada)

Joint work of Andreas Abel; Andrew Cave; Brigitte Pientka; Anton Setzer; David Thibodeau

Coinduction is a key proof technique to establish properties about systems that continue to run and produce results (i.e. network or communications protocols, I/O interaction, data streams, or processes) . Yet, mechanizing coinductive proofs about formal systems and representing, generating and manipulating such proofs remains challenging. In this talk, we develop the idea of programming coinductive proofs dual to the idea of programming inductive proofs. Unlike properties about finite data which can be defined by constructing a derivation, properties about infinite data can be described by the possible observations we can make. Dual to pattern matching, a tool for analyzing finite data, we develop the concept of copattern matching, which allows us to describe properties about infinite data. This leads to a symmetric proof language where pattern matching on finite and infinite data can be mixed.

3.20 Language-based Verification of Untyped Expressions

Ruzica Piskac (Yale University, US)

Software failures resulting from configuration errors have become commonplace as modern software systems grow increasingly large and more complex. The lack of language constructs in configuration files, such as types and grammars, has directed the focus of a configuration file verification towards building post-failure error diagnosis tools. In addition, the existing tools are generally language specific, requiring the user to define at least a grammar for the language models and explicit rules to check.

In this talk, we outline a framework which analyzes datasets of correct configuration files and derives rules for building a language model from the given dataset. The resulting language model can be used to verify new configuration files and detect errors in them. Our proposed framework is highly modular, does not rely on the system source code, and can be applied to any new configuration file type with minimal user input.

3.21 Program Synthesis from Refinement Types

Nadia Polikarpova (MIT – Cambridge, US)

Joint work of Ivan Kuraj; Nadia Polikarpova; Armando Solar-Lezama

The key to scalable program synthesis is modular verification: the better a specification for a program can be broken up into independent specifications for its components, the fewer combinations of components the synthesizer has to consider, leading to a combinatorial reduction in the size of the search space. This talk will present Synquid: a synthesizer that takes advantage of the modularity offered by type-based verification techniques to efficiently generate recursive functional programs that provably satisfy a given specification in the form of a refinement type.

We have evaluated Synquid on a large set of synthesis problems and found that it exceeds the state of the art in terms of both scalability and usability. Synquid was able to synthesize more complex programs than those reported in prior work (for example, various sorting algorithms, operations on balanced trees). It was also able to handle most of the benchmarks tackled by existing tools, often starting from a significantly more concise and intuitive user input. Moreover, due to automatic refinement discovery through a variant of liquid type inference, our approach fundamentally extends the class of programs for which a provably correct implementation can be synthesized without requiring the user to provide full inductive invariants.

3.22 Demand Driven Analysis For Functional Programs

Scott Smith (Johns Hopkins University – Baltimore, US)

Joint work of Zachary Palmer, Scott Smith
Main reference Z. Palmer, S. Smith, "Higher-Order Demand-Driven Program Analysis", in Proc. of the 30th Europ. Conf. on Object-Oriented Programming (ECOOP'16), LIPIcs, Vol. 56, pp. 19:1–19:25, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.
URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.19

We explore a novel approach to higher-order program analysis that brings ideas of on-demand lookup from first-order CFL-reachability program analyses to functional programs. The analysis needs to produce only a control-flow graph; it can derive all other information including values of variables directly from the graph. Several challenges had to be overcome, including how to build the control-flow graph on-the-fly and how to deal with nonlocal variables in functions. The resulting analysis is flow- and context-sensitive with a provable polynomial-time bound.

3.23 Equations: A toolbox for function definitions in Coq

Matthiew Sozeau (Université Paris Diderot – Paris, France)

We present a compiler for definitions made by pattern matching on inductive families in the Coq system. It allows to write structured, recursive dependently-typed functions, automatically find their realization in the core type theory and generate proofs to ease reasoning on them. The high-level interface allows to write dependently-typed functions on inductive families in a style close to Agda or Epigram, while their low-level implementation is accepted by the vanilla core type theory of Coq. This setup uses the smallest trusted code base possible and additional tools are provided to maintain a high-level view of definitions.

3.24 Temporal Verification of Higher-Order Functional Programs

Tachio Terauchi (JAIST – Japan)

We present an automated approach to verifying arbitrary omega-regular properties of higher-order functional programs. Previous automated methods proposed for this class of programs could only handle safety properties or termination, and our approach is the first to be able to verify arbitrary omega-regular liveness properties. Our approach is automata-theoretic, and extends our recent work on binary-reachability-based approach to automated termination verification of higher-order functional programs to fair termination published in ESOP 2014. In that work, we have shown that checking disjunctive well-foundedness of (the transitive closure of) the "calling relation" is sound and complete for termination. The extension to fair termination is tricky, however, because the straightforward extension that checks disjunctive well-foundedness of the fair calling relation turns out to be unsound, as we shall show in the paper. Roughly, our solution is to check fairness on the transition relation instead of the calling relation, and propagate the information to determine when it is necessary and sufficient to check for disjunctive well-foundedness on the calling relation. We prove that our approach is sound and complete. We have implemented a prototype of our approach, and confirmed that it is able to automatically verify liveness properties of some non-trivial higher-order programs.

3.25 Occurrence typing modulo theories

Sam Tobin-Hochstadt (Indiana University – Bloomington, US)

Occurrence typing has been successful in enabling Typed Racket to handle a wide variety of existing Racket idioms. In this talk, I present a new extension, adding dependent refinement types parameterized over arbitrary solvers to Typed Racket.

Dependent refinement types allow Typed Racket programmers to express rich type relationships, ranging from data structure invariants such as red-black tree balance to preconditions such as vector bounds. Refinements allow programmers to embed the propositions that occurrence typing in Typed Racket already reasons about into their types. Further, extending occurrence typing to refinements allows us to make the underlying formalism simpler and more powerful.

3.26 Refinement Caml: A Refinement Type Checking and Inference Tool for OCaml

Hiroshi Unno (Tsukuba University – Japan)

We will demonstrate Refinement Caml (RCaml), a fully-automated path-sensitive verification tool for the OCaml functional language based on refinement type checking and inference. RCaml supports advanced language features such as algebraic data structures and higher-order recursive functions. RCaml can solve various program analysis problems formulated as refinement type inference problems, including static assertion checking, termination and non-termination analysis, precondition inference, relational verification, and symbolic game solving. RCaml first reduces these problems into constraint solving problems, where the constraints are expressed by Horn clauses with predicate variables that are placeholders for preconditions, postconditions, safe inductive invariants, and well-founded recursion relations of the original program. RCaml then solves the generated constraints by using invariant and ranking function synthesis techniques.

3.27 Contract Verification and Refutation

David Van Horn (University of Maryland – College Park, USA)

In this talk, I'll present a new approach to automated reasoning about higher- order programs by endowing symbolic execution with a notion of higher-order, symbolic values. Our approach is sound and relatively complete with respect to a first-order solver for base type values. Therefore, it can form the basis of automated verification and bug-finding tools for higher-order programs. To validate our approach, we use it to develop and evaluate a system for verifying and refuting behavioral software contracts of components in a functional language, which we call soft contract verification. In doing so, we discover a mutually beneficial relation between behavioral contracts and higher-order symbolic execution.

3.28 Refinement Types for Haskell

Niki Vazou (University of California – San Diego, US)

License 🅭 Creative Commons BY 3.0 Unported license
© Niki Vazou
Joint work of Alexander Bakst; Ranjit Jhala; Eric L. Seidel; Niki Vazou
Main reference N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, S. Peyton-Jones, "Refinement Types For Haskell", Manuscript.
URL http://goto.ucsd.edu/~nvazou/refinement_types_for_haskell.pdf
Main reference N. Vazou, A. Bakst, R. Jhala, "Bounded refinement types", in Proc. of the 20th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'15), pp. 48–61, ACM, 2015; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2784731.2784745
URL https://ranjitjhala.github.io/static/bounded_refinement_types.pdf

Haskell has many delightful features, perhaps the most beloved of which is its type system which allows developers to specify and verify a variety of program properties at compile time. However, many properties, typically those that depend on relationships between program values are impossible, or at the very least, cumbersome to encode within Haskell's type system.

Liquid types enable the specification and verification of value-dependent properties by extending Haskell's type system with logical predicates drawn from efficiently decidable logics.

In this talk, we will start with a high level description of Liquid Types. Next, we will present an overview of LiquidHaskell, a liquid type checker for Haskell. In particular, we will describe the kinds of properties that can be checked, ranging from generic requirements like totality (will 'head' crash?) and termination (will 'mergeSort' loop forever?), to application specific concerns like memory safety (will my code SEGFAULT?) and data structure correctness invariants (is this tree BALANCED?).

3.29 Lazy staged binary decision diagrams

Wouter Swierstra (Utrecht University, NL)

License 🅭 Creative Commons BY 3.0 Unported license
© Wouter Swierstra
Joint work of Joao Paulo Pizani Flor; Rob Spoel; Wouter Swierstra

This talk reviews the proof-by-reflection method of proof automation. By exploiting the computational nature of type theory, we can implement a verified decision procedure for a specific domain, rather than write manual proof terms. We would like to use this to implement a decision procedure on boolean expressions. The usual techniques to do so, binary decision diagrams, rely on manipulating pointers in memory – which does not work well with most proof assistants based on type theory. Instead, we propose to investigate how we might be able to create the desired data structures in memory using metaprogramming.

3.30 Verification by Optimization: Two Approaches to Manifest Contracts

Michael Greenberg (Pomona College – Claremont, US)

Contract systems can be used for program verification: if you can optimize away the contract checks, you have proved the program correct!

There are two approaches in the literature: subtyping and static analysis. The subtyping approach (typical in the manifest setting, and due to Flanagan) removes upcasts, casts from a subtype to a super type. The static analysis approach (typical in the latent setting, and due most immediately to Van Horn, though many others have written on similar topics before) builds an abstract model of which values each variable can hold – if the set of values all pass a given contract, that contract can be eliminated.

Is one "better" than the other? Can we use both in the same setting? Do they optimize away different kinds of contracts?

In this brief talk, I lay out the differences between the two and propose a "non-disjointness" judgment for determining when to reject a program because of a bad cast.

4 Breakout Sessions

In addition to the formal talks, we had breakout sessions on the following topics:

- **Integrating formal methods tools in the curriculum.** In this session, we discussed a variety of topics ranging from current best exemplars of classes and textbooks for formal methods, what makes for a good class or text, and regular classes (e.g. operating systems or compilers) that could be extended with Formal Methods module.
- **Hands on Tool Tutorials.** In this session, different participants gave short demonstrations of their tools and then encouraged others to use them to carry out various tasks or work through tutorials. These tools included: Agda, Dafny, F^*, Haskell, LiquidHaskell, Racket, RCaml, Synquid and Why3.
- **User Interaction** In this session, the participants identified several key challenges that must be addressed to improve the user experience for formal tools, as well as new modes of using the tools, e.g. not just for verification but to synthesize programs.
- **Types and Effects** In this session, the participants discussed recent advances in how to track effects, and different applications of effect systems.

Participants

- Andreas Martin Abel
Chalmers UT – Göteborg, SE
- Amal Ahmed
Northeastern University – Boston, US
- Andrew W. Appel
Princeton University, US
- Lennart Augustsson
Standard Chartered Bank – London, GB
- Edwin Brady
University of St. Andrews, GB
- Iavor Diatchki
Galois – Portland, US
- Richard A. Eisenberg
University of Pennsylvania – Philadelphia, US
- Jean-Christophe Filliâtre
CNRS & University Paris Sud, FR
- Cormac Flanagan
University of California – Santa Cruz, US
- Marco Gaboardi
SUNY – Buffalo, US
- Deepak Garg
MPI-SWS – Saarbrücken, DE
- Michael Greenberg
Pomona College – Claremont, US
- Reiner Hähnle
TU Darmstadt, DE
- Cătălin Hriţcu
INRIA – Paris, FR
- Suresh Jagannathan
Purdue University – West Lafayette, US
- Ranjit Jhala
University of California – San Diego, US
- Gowtham Kaki
Purdue University – West Lafayette, US
- Gabriele Keller
UNSW – Sydney, AU
- Naoki Kobayashi
University of Tokyo, JP
- Ekaterina Komendantskaya
University of Dundee, GB
- Martin Lange
Universität Kassel, DE
- K. Rustan M. Leino
Microsoft Corporation – Redmond, US
- Conor McBride
University of Strathclyde – Glasgow, GB
- Jan Midtgaard
Technical University of Denmark – Lyngby, DK
- Chih-Hao Luke Ong
University of Oxford, GB
- Dominic Orchard
University of Cambridge, GB
- Brigitte Pientka
McGill Univ. – Montreal, CA
- Ruzica Piskac
Yale University, US
- Nadia Polikarpova
MIT – Cambridge, US
- Scott Smith
Johns Hopkins University – Baltimore, US
- Matthieu Sozeau
University Paris-Diderot, FR
- Wouter Swierstra
Utrecht University, NL
- Tachio Terauchi
JAIST – Ishikawa, JP
- Sam Tobin-Hochstadt
Indiana University – Bloomington, US
- Hiroshi Unno
University of Tsukuba, JP
- David Van Horn
University of Maryland – College Park, US
- Niki Vazou
University of California – San Diego, US
- Stephanie Weirich
University of Pennsylvania – Philadelphia, US
- Nobuko Yoshida
Imperial College London, GB