JFP **35**, e2, 11 pages, 2025.

doi:10.1017/S0956796824000170

PhD Abstracts

GRAHAM HUTTON
University of Nottingham, UK
(*e-mail:* graham.hutton@nottingham.ac.uk)

Many students complete PhDs in functional programming each year. As a service to the community, twice per year the Journal of Functional Programming publishes the abstracts from PhD dissertations completed during the previous year.

The abstracts are made freely available on the JFP website, i.e. not behind any paywall. They do not require any transfer of copyright, merely a license from the author. A dissertation is eligible for inclusion if parts of it have or could have appeared in JFP, that is, if it is in the general area of functional programming. The abstracts are not reviewed.

We are delighted to publish ten abstracts in this round and hope that JFP readers will find many interesting dissertations in this collection that they may not otherwise have seen. If a student or advisor would like to submit a dissertation abstract for publication in this series, please contact the series editor for further details.

Graham Hutton
PhD Abstract Editor

Gradual Intersection Types

PEDRO JORGE FERNANDES ÂNGELO
Universidade do Porto, Portugal

Date: June 2024; Advisor: Mário Florido
URL: https://tinyurl.com/fwf5wv7f

The aim of this thesis is the integration of gradual typing into systems with intersection types. We focus on two target calculi for this analysis: the λ-calculus and the core object calculus Featherweight Java, or rather, its extension with interfaces, λ-expressions and intersection types.

During the course of the research presented therein, several challenges arose. Most of these are due to the intrinsic differences between intersection types and gradual typing, both as opposing requirements of implementation. These requirements can be summarized as follows: gradual typing is defined for the Church-style λ-calculi, whereas intersection types were originally intended, and still predominantly defined, as a type assignment system for the Curry-style λ-calculus. Furthermore, type inference algorithms for both gradual typing and intersection types follow opposed approaches when it comes to assigning types to occurrences of variables.

In this thesis, we show how these differences can be reconciled and how the challenges were overcome. We also show how the insights gleaned from the work with λ-calculus can be applied to object oriented languages, such as Java. The contributions in this thesis are:

- a new type inference algorithm for rank 2 intersection types, which is both sound and complete with respect to an intersection type system;
- a type system integrating gradual typing with intersection types, shown to be type safe and to satisfy most of the correctness criteria for gradual typing, including the gradual guarantee;
- a sound and complete type inference algorithm for rank 2 gradual intersection types;
- an extension to Featherweight Java with interfaces, λ-expressions, intersection types and the dynamic type, providing a first step towards an integration of gradual typing with Java.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press

Tool-Driven Quality Assurance for Functional Programming and Machine Learning

LEONHARD APPLIS
Technical University of Delft, The Netherlands

Date: October 2024; Advisor: Arie van Deursen and Annibale Panichella
URL: https://tinyurl.com/2vrt9zdu

Software Engineering heavily relies on tools, and functional programming is no exception to this. Yet, tooling efforts are often concentrated on the compiler and types, while other fields such as debugging, benchmarking and maintenance are slightly underrepresented. This work first presents a new dataset of Haskell Bugs (HasBugs) and asserts how they compare and differ to other established work from e.g. Java, followed by an investigation of a prominent feature of Haskell: Lazy Evaluation. While usually benign, it can lead to unique issues due to a difference in call- and evaluation-sequence that is not well represented in error messages. To this end, we enrich HPCs coverage from calls to calls and evaluations, which greatly benefits locating certain errors, such as 'NonExhaustivePatternMatches'. In later chapters, the rich type system of Haskell and its availability through GHC are used to bring classic automated software engineering (ASE) approaches like automated program repair (APR) and spectrum-based fault localisation (SBFL) to Haskell. For automated program repair we utilise the type system to source automatically compiling repairs from typed holes and investigate the randomised nature of properties to avoid overfitting of repair-candidates. Spectrum-based fault localisation and common formulas were reasonably effective to be useful, however introducing available information from ASTs and types in addition to the coverage did not immediately improve over existing approaches. This thesis takes the first steps to introduce information about the program, its types and tests into ASE to both eliminate false-friends and benefit performance by reducing search spaces. There are two reasons why we should care about better tooling: First, many issues are shared with other programming paradigms, and transplanting their solutions can free developer capacities. Second, there are unique problems within Haskell, and having the right tool makes fixing them much easier. With better tools we have more time to do what we like most: Enjoy functional programming.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press

Generic Bidirectional Typing in a Logical Framework for Dependent Type Theories

THIAGO FELICISSIMO
Université Paris-Saclay, France

Date: September 2024; Advisor: Frédéric Blanqui and Gilles Dowek
URL: https://tinyurl.com/mtwytzdb

Dependent type theories are formal systems that can be used both as programming languages and for the formalization of mathematics, and constitute the foundation of popular proof assistants such as Coq and Agda. In order to unify their study, Logical Frameworks (LFs) provide a unified meta-language for defining such theories in which various universal notions are built in by default and metatheorems can be proven in a theory-independent way. This thesis focuses on LFs designed with implementation in mind, with the goal of providing generic type-checkers. Our main contribution is a new such LF which allows for representing type theories with their usual non-annotated syntaxes. The key to allowing the removal of annotations without jeopardizing decidability of typing is the integration of bidirectional typing, a discipline in which the typing judgment is decomposed into inference and checking modes. While bidirectional typing has been well known in the literature for quite some time, one of the central contributions of our work is that, by formulating it in an LF, we give it a generic treatment for all theories fitting our framework. Our proposal has been implemented in the generic type-checker BiTTs, allowing it to be used in practice with various theories. In addition to our main contribution, we also advance the study of Dedukti, a sibling LF of our proposed framework. First, we revisit the problem of showing that theories are correctly represented in Dedukti by proposing a methodology for encodings which allows for showing their conservativity easily. Furthermore, we demonstrate how Dedukti can be used in practice as a tool for translating proofs by proposing a transformation for sharing proofs with predicative systems. This transformation has allowed for the translation of proofs from Matita to Agda, yielding the first-ever Agda proofs of Fermat's Little Theorem and Bertrand's Postulate.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press

Synthesis and Repair for Functional Programming: A Type- and Test-Driven Approach

MATTHÍAS PÁLL GISSURARSON
Chalmers University of Technology, Sweden

Date: August 2024; Advisor: David Sands
URL: https://tinyurl.com/3bv2m7y6

Modern programs in languages like Haskell include a lot of information beyond what is required for compilation. This includes unit tests, property-based tests, and type annotations more specific than those necessary to resolve ambiguity. This additional specification is usually only used for post-compilation verification by running the tests to verify that the code-as-written matches the specification the types and properties provide.

In this thesis, we explore ways of going beyond verification, and how this additional information can aid the developer during development. This can be done in multiple ways, for example, by helping the programmer write an implementation that matches the specification, by helping them track down the source of a bug in the implementation, and automatically repairing an implementation that does not match the specification.

In the first part, I explore the integration of program synthesis into GHC compiler error messages using typed-hole suggestions to aid completion of partial programs during development. In the second part, we present PropR, an automatic repair tool. PropR is based on type-driven synthesis, guided by property-based testing and fault localization in conjunction with genetic algorithms. A rich specification is required for these approaches to be effective. This motivates the third part of this thesis, where we present Spectacular, a specification synthesis tool. Spectacular uses ECTA-based synthesis to automatically infer properties of programs, letting us bootstrap specifications from previous versions.

In the fourth and fifth part of this thesis, we present the lightweight trace-based and spectrum-based fault localization tools CSI: Haskell and TastySpectrum respectively, and explore how we can localize program faults and find likely sources of a bug.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press

Property-Based Testing for the People

HARRISON GOLDSTEIN
University of Pennsylvania, USA

Date: August 2024; Advisor: Benjamin C. Pierce
URL: https://tinyurl.com/528pvryc

Software errors are expensive and dangerous. Best practices around testing help to improve software quality, but not all testing tools are created equal. In recent years, automated testing, which helps to save time and avoid developers' blind-spots, has begun to improve this state of affairs.

In particular, *property-based testing* is a powerful automated testing technique that gives developers some of the power of formal methods without the high cost. PBT is effective at finding important bugs in real systems, and it has been established as a go-to technique for testing in some localized developer communities.

To bring the power of PBT to a larger demographic, I shift focus to PBT *users*. My work is motivated by conversations with real developers, accentuating the benefits that they get from PBT and reducing the drawbacks. My work begins with *Property-Based Testing in Practice*. This user study establishes a rich set of observations about PBT's use in practice, along with a wide array of ideas for future research that are motivated by the needs of real PBT users. The rest of the work in the dissertation is informed by these results. One critical observation is that PBT users struggle with the random data generation that is key to PBT's operation. To address this problem, I establish a new foundation for random data generators in *Parsing Randomness* and extend that foundation to be more flexible and powerful in *Reflecting on Randomness*. These projects contribute new algorithms that increase developers' leverage during testing while decreasing developer effort. I also observed that PBT users are not always good at evaluating whether their testing was effective. In *Understanding Randomness*, I establish a new PBT paradigm that gives developers critical insights into their tests' performance and enables new ways of interacting with a PBT system. The Tyche interface developed in that project is now an open source tool with real-world users.

By blending tools from programming languages, human-computer interaction, and software engineering, my work increases the reach and impact of PBT and builds a foundation for a future with better software assurance.

Types With Extra Structure: Predicates, Equations, Composition

BRANDON HEWER
University of Nottingham, UK

Date: September 2024; Advisor: Graham Hutton
URL: https://tinyurl.com/nsnenv7a

Intuitionistic type theory was first introduced by Martin Lof as a foundation for constructive mathematics and also serves as a dependently typed programming language. Dependent types provide us with a framework to reason about and guide the construction of programs by specifying both their structure and properties in a manner that can be automatically verified by a type-checker.

A ubiquitous pattern that arises in the formulation of dependent type abstractions involves equipping an underlying type, which captures the general form of a program, with extra structure that captures the program's properties. Two such type abstractions include subtypes in which a type is equipped with a predicate over its values and quotient types in which a type is equipped with equations over its values. While subtypes have found much practical use in general purpose programming, quotient types have not seen many applications outside of proof assistants. Two key obstacles to the wider adoption of quotient types include an absence of practical demonstrations of their applications to general purpose programming and the significant burden of proof-obligations that arises from their use.

In this thesis, we introduce three new applications of type theoretic concepts that involve equipping types with extra structure. Firstly, we introduce a new practical application for higher-inductive types whereby they are used to encode subtypes to provide fine-grained control over the reduction behaviour of terms. Our second key contribution is the extension of a liquid type system to include a class of quotient types for which the necessary proof-obligations are decidable by an SMT-solver. This work is accompanied by a practical demonstration in the form of Quotient Haskell, which was developed as an extension to Liquid Haskell. Finally, we present a constructive theory of operads, which were first introduced by Peter May to describe composable algebraic structures in symmetric monoidal structures. Intuitively, an operad can be understood as a finite family of types equipped with a well-behaved notion of composition. We demonstrate how a theory of operads in homotopy type theory gives rise to a generic framework for reasoning about collections of operations.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press

Program Synthesis from Linear and Graded Types

JACK OLIVER HUGHES
University of Kent, UK

Date: November 2024; Advisor: Dominic Orchard
URL: https://tinyurl.com/yeyzxp4f

Graded types are a class of resourceful types which allow for fine- grained quantitative reasoning about data-flow in programs. Tracing their roots from linear types, the use of resource annotations (or grades) on data, allows a programmer to express structural or semantic properties of their program at the type level. Such systems have become increasingly popular in recent years, mainly for the expressive power that they offer to programmers; judicious use of grades in type specifications significantly reduces the number of typeable programs. These additional constraints on types lend themselves naturally to type-directed program synthesis, which leverage the information provided by types to prune ill-resourced programs from the search space of candidate programs. In synthesis, this grade information can be exploited to constrain the search space of programs even further than in standard type systems. We present an approach to program synthesis for linear and graded type systems, where grades form an arbitrary pre-ordered semiring. Harnessing this grade information in synthesis is non-trivial, and we explore the issues involved in designing and implementing a resource-aware program synthesis tool, culminating in an efficient and expressive program synthesis tool for the research programming language Granule, which uses a graded type system. We show that by harnessing grades in synthesis, the majority of our benchmarking synthesis problems (many of which involve recursive functions over recursive ADTs) require less exploration of the synthesis search space than a purely type-driven approach and with fewer needed input-output examples. Our type-and-graded-directed approach is demonstrated in the Granule but we also adapt it for synthesising Haskell programs that use GHC's Linear Types extension, demonstrating the versatility of our approach to resourceful program synthesis.

Lightweight Approaches to the Verification of Functional Programs

EDDIE JONES
University of Bristol, UK

Date: March 2024; Advisor: Steven Ramsay
URL: https://tinyurl.com/yrpexayw

The constraints of pure functional programs are often applauded for the resulting safety and correctness guarantees. It is also claimed that these programs are easier to reason about and, therefore, verify. Despite being taken as fact within the community, the availability of effective verification tools tells a different story. This thesis focuses on two verification problems specific to functional programs — pattern-match safety and functional correctness. We develop two automated, lightweight verification tools with a focus on performance.

The first problem is to verify that a given functional program does not crash due to inexhaustive pattern-matching expressions in a function's definition. To this end, we present a refinement type system with a restricted form of structural subtyping and environment-level intersection. We describe a fully automated, sound and complete type inference procedure for this system which, under reasonable assumptions, is worst-case linear-time in the size of the program. Compositionality is essential to obtaining this complexity guarantee but is only enabled by the novel restriction we place on refinement types.

Other than expressive type systems, pure functional programs naturally lend themselves to equational specifications. These specifications are a desirable target for an automated verification tool because they are immediately accessible to the average programmer. Nevertheless, such a tool must tackle the thorny issue of proof by induction when verifying recursive programs over algebraic datatypes. We propose a new cyclic proof system that is well-adapted to equational reasoning over inductively defined datatypes. The key to our system is the way in which cyclic proofs and equational reasoning are mediated through the use of contextual substitution as a cut-like rule. We outline a performant proof search algorithm that relies on a number of supporting theoretical developments, including an alternative, incremental technique for checking the correctness of a candidate proof.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press

Automatic Differentiation via Effects and Handlers

JESSE AARON SIGAL
University of Edinburgh, UK

Date: June 2024; Advisor: Chris Heunen, James Cheney and Ian Stark
URL: https://tinyurl.com/523t4t62

Machine learning, artificial intelligence, scientific modelling, information analysis, and other data heavy fields have driven the demand for tools that enable derivative based optimization. Automatic differentiation is a family of algorithms used to calculate the derivatives of programs with only a constant factor slowdown. There are many implementation strategies, some built into a language and some outside of it, and there are many different members of the family. The utility of automatic differentiation makes it worthwhile to implement it in as many languages as possible.

Effects and handlers are a powerful control flow construct in programming languages based upon delimited continuations. They are a structured method of including side effects into programs, and have found many uses including nondeterminism, state management, and concurrency. Effects and handlers excel in facilitating non-local control flow and also provide methods of abstracting and composing effects. Mainstream programming languages are increasingly incorporating effects and handlers, notably OCaml and WebAssembly.

We show that effects and handlers are well-suited for implementing automatic differentiation algorithms while maintaining the desirable asymptotic efficiency. In particular, effects and handlers allow for succinctness in the presence of complex control flow. On a practical level, we implement eight automatic differentiation algorithms in four languages with effects and handlers. The implementations range from standard AD algorithms such as forward mode and continuation-based reverse mode, to more advanced modes such as checkpointed reverse mode. We benchmark the standard modes to empirically show that we can reach the correct asymptotic complexity.

Furthermore, we build up a mathematical framework in which to prove correctness of selected standard modes. To do so, we extend the set-theoretic denotational semantics of a simple effect and handler language to a category-theoretic semantics. We then describe how to perform a generalized proof by logical relations in this setting, and identify sufficient conditions for our proof method to apply. Equipped with our conditions, we show that diffeological spaces (a generalization of Euclidean spaces) admit proof by logical relations. Ultimately, this enables us to prove our implementations of forward mode and continuation reverse mode correct.

A Framework for Semiring-Annotated Type Systems

JAMES WOOD
University of Strathclyde, UK

Date: October 2024; Advisor: Robert Atkey
URL: https://tinyurl.com/yckxnhjb

The use of proof assistants as a tool for programming language theorists is becoming ever more practical and widespread. There is a range of satisfactory implementations of simply typed calculi in proof assistants based on dependent type theory.

In this thesis, I extend an account of Simply Typed λ-calculus so as to be able to represent and reason about calculi whose variables have restricted usage patterns. Examples of such calculi include a logic with an S4 □-modality, in which certain variables cannot be used "inside" a box (□); and Linear Logic, in which linear variables have to be used exactly once. While there are existing implementations of some of these calculi in proof assistants, many of these implementations share little with the best presentations of simply typed calculi without variable usage restrictions, and thus end up being poorly understood or suboptimal in facilitating mechanised reasoning.

Concretely, the main result of this thesis is a framework for representing and reasoning about a wide range of calculi with restricted variable usage. All of these calculi support novel simultaneous renaming and substitution operations. Furthermore, I provide several other examples of generic and specific programs facilitated by the framework. All of this work is implemented in the proof assistant Agda.

https://doi.org/10.1017/S0956796824000170 Published online by Cambridge University Press