

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

Computational Semantics of Cartesian Cubical Type Theory

CARLO ANGIULI
Carnegie Mellon University, USA

Date: September 2019; Advisor: Robert Harper
URL: <https://tinyurl.com/tk87llc>

Dependent type theories are a family of logical systems that serve as expressive functional programming languages and as the basis of many proof assistants. In the past decade, type theories have also attracted the attention of mathematicians due to surprising connections with homotopy theory; the study of these connections, known as *homotopy type theory*, has in turn suggested novel extensions to type theory, including higher inductive types and Voevodsky’s univalence axiom. However, in their original axiomatic presentation, these extensions lack computational content, making them unusable as programming constructs and unergonomic in proof assistants.

In this dissertation, we present *Cartesian cubical type theory*, a univalent type theory that extends ordinary type theory with interval variables representing abstract hypercubes. We justify Cartesian cubical type theory by means of a *computational semantics* that generalizes Allen’s semantics of Nuprl to Cartesian cubical sets. Proofs in our type theory have computational content, as evidenced by the canonicity property that all closed terms of Boolean type evaluate to true or false. It is the second univalent type theory with canonicity, after the De Morgan cubical type theory of Cohen, Coquand, Huber, and Mörtberg, and affirmatively resolves an open question of whether Cartesian interval structure constructively models univalent universes.

Learning Proof Search in Proof Assistants

MICHAEL FÄRBER
University of Innsbruck, Austria

Date: November 2018; Advisor: Cezary Kaliszyk
URL: <https://tinyurl.com/wbpguu6>

Proof assistants are programs that verify the correctness of formal proofs. They can increase the confidence in results from domains such as mathematics, informatics, physics, and philosophy. However, it requires extensive labour and expertise to write proofs accepted by proof assistants. In this thesis, we improve proof automation in proof assistants.

Automated theorem provers are programs that search for proofs automatically. Our goal is to find proofs in proof assistants using automated theorem provers. However, this is not directly possible when the logic of an automated theorem prover and that of a proof assistant differ.

In this case, the integration of the automated theorem prover into the proof assistant requires the translation of statements to the logic of the automated theorem prover and the translation of proofs to the logic of the proof assistant. To restrict the search space of the automated theorem prover, only a selection of facts relevant to the current conjecture is translated. The success rate of the automatic proof search in proof assistants depends on the various translations, the selection of relevant facts as well as on the automated theorem prover itself.

We improve the integration of automated theorem provers into proof assistants. Among others, we learn from previous proofs to select relevant facts as well as to guide automated theorem provers to make good decisions. Furthermore, we create automated proof translations for several automated theorem provers for which such a translation was not previously available. Finally, we evaluate different implementation methods, such as continuation-passing style and lazy lists, to create efficient and compact automated theorem provers in functional languages. All the implementations in the thesis are written in functional languages (mostly OCaml and Haskell), and are publicly available.

Our work increases the success ratio of proof search in proof assistants.

Typed Concurrent Functional Programming with Channels, Actors, and Sessions

SIMON FOWLER
University of Edinburgh, UK

Date: July 2019; Advisor: Sam Lindley and Philip Wadler
URL: <https://tinyurl.com/volljq8>

The age of writing single-threaded applications is over: to develop scalable applications, developers must make use of concurrency and parallelism. Unfortunately, writing concurrent code is prone to issues such as race conditions and deadlocks, and moving to the distributed setting introduces the possibility of failure.

To cope with the problems of concurrent programming, language designers have proposed *communication-centric* programming languages, which eschew shared memory in favour of message passing. The focus of this thesis is on *typed* communication-centric functional programming languages, using type systems to provide static guarantees about concurrent programs. We investigate two strands of work: the relationship between typed channel- and actor-based languages, and the integration of asynchrony, exception handling, and session types.

In the first strand, we investigate two particular subclasses of communication-centric languages: channel-based languages such as Go, and actor-based languages, such as Erlang. We distil the essence of the languages into two concurrent lambda-calculi: lambda-ch for simply-typed channels, and lambda-act for simply-typed actors, and provide type- and semantics-preserving translations between them. In doing so, we clear up confusion between the two models, give theoretical foundations for type-parameterised actors, and provide a theoretical grounding for frameworks which emulate actors in channel-based languages. By extending the core calculi, we note that synchronisation simplifies the translation from channels into actors, and show an encoding of Erlang's selective receive mechanism.

In the second strand, we integrate session types, asynchrony, and exception handling in a functional programming language. Session types are a type system for channel endpoints, allowing protocol conformance to be checked statically. We provide the first integration of exception handling and asynchronous session types in a core functional language, *Exceptional GV*, proving that it satisfies preservation, global progress, and that it is confluent and terminating. We demonstrate the applicability of the approach by extending the Links web programming language with exception handling, providing the first implementation of exception handling in the presence of session types in a functional language. As a result, we show the first application of session types to web programming, providing examples including a two-factor authentication workflow and a chat application.

Advanced Logical Type Systems for Untyped Languages

ANDREW M. KENT
Indiana University, USA

Date: October 2019; Advisor: Sam Tobin-Hochstadt
URL: <https://tinyurl.com/uf7p8gh>

Type systems with occurrence typing—the ability to refine the type of terms in a control flow sensitive way—now exist for nearly every untyped programming language that has gained popularity. While these systems have been successful in type checking many prevalent idioms, most have focused on relatively simple verification goals and coarse interface specifications. We demonstrate that such systems are naturally suited for combination with more advanced type theoretic concepts—specifically refinement types and semantic subtyping—with both formal mathematical models and experiences reports from implementing such systems at scale.

Non-Reformist Reform for Haskell Modularity

SCOTT KILPATRICK
Universität des Saarlandes, Germany

Date: October 2019; Advisor: Derek Dreyer
URL: <https://tinyurl.com/qs37gt9>

Module systems like that of Haskell permit only a weak form of modularity in which module implementations depend directly on other implementations and must be processed in dependency order. Module systems like that of ML, on the other hand, permit a stronger form of modularity in which explicit interfaces express assumptions about dependencies and each module can be typechecked and reasoned about independently.

In this thesis, I present Backpack, a new language for building separately-typecheckable packages on top of a weak module system like Haskell's. The design of Backpack is the first to bring the rich world of type systems to the practical world of packages via mixin modules. It's inspired by the MixML module calculus of Rossberg and Dreyer but by choosing practicality over expressivity Backpack both simplifies that semantics and supports a flexible notion of applicative instantiation. Moreover, this design is motivated less by foundational concerns and more by the practical concern of integration into Haskell. The result is a new approach to writing modular software at the scale of packages.

The semantics of Backpack is defined via elaboration into sets of Haskell modules and binary interface files, thus showing how Backpack maintains interoperability with Haskell while retrofitting it with interfaces. In my formalization of Backpack I present a novel type system for Haskell modules and I prove a key soundness theorem to validate Backpack's semantics.

Type Systems for Systems Types

LIAM O'CONNOR

University of New South Wales, Australia

Date: October 2019; Advisor: Gabriele Keller, Christine Rizkallah and Gernot Heiser

URL: <https://tinyurl.com/uns8byh>

This thesis presents a framework aimed at significantly reducing the cost of proving functional correctness for low-level operating systems components, designed around a new programming language, Cogent. This language is total, polymorphic, higher-order, and purely functional, including features such as algebraic data types and type inference. Crucially, Cogent is equipped with a uniqueness type system, which eliminates the need for a trusted runtime or garbage collector, and allows us to assign two semantics to the language: one imperative, suitable for efficient C code generation; and one functional, suitable for equational reasoning and verification. We prove that the functional semantics is a valid abstraction of the imperative semantics for all well-typed programs. Cogent is designed to easily interoperate with existing C code, to enable Cogent software to interact with existing C systems, and also to provide an escape hatch of sorts, for when the restrictions of Cogent's type system are too onerous. This interoperability extends to Cogent's verification framework, which composes with existing C verification frameworks to enable whole systems to be verified.

Cogent's verification framework is based on certifying compilation: For a well-typed Cogent program, the compiler produces C code, a high-level representation of its semantics in Isabelle/HOL, and a proof that the C code correctly refines this embedding. Thus one can reason about the full semantics of real-world systems code productively and equationally, while retaining the interoperability and leanness of C. The compiler certificate is a series of language-level proofs and per-program translation validation phases, combined into one coherent top-level theorem in Isabelle/HOL.

To evaluate the effectiveness of this framework, two realistic file systems were implemented as a case study, and key operations for one file system were formally verified on top of Cogent specifications. These studies demonstrate that verification effort is drastically reduced for proving higher-level properties of file system implementations, by reasoning about the generated formal specification from Cogent, rather than low-level C code.

*Specification and Verification of
Actor Protocols with Finite-State Machines*

JONATHAN SCHUSTER
Northeastern University, USA

Date: August 2019; Advisor: Olin Shivers
URL: <https://tinyurl.com/qssmd5>

Many programmers use the actor model to build distributed systems. The communication aspects of such systems are notoriously hard to implement correctly, however, leading programmers to spend more time debugging protocol implementations and less time focusing on application logic. Furthermore, the common approach of specifying a protocol as a finite-state machine and verifying that the program implements this protocol is insufficient, because standard FSMs do not account for the dynamic, evolving communication topologies in actor programs.

To address this problem, this dissertation defines a specification language that augments finite-state machines with the ability to describe address-passing aspects of actor protocols. Additionally, the dissertation develops a series of proof techniques for such specifications, as well as a model-checking algorithm that verifies whether a program conforms to its specification. When applied to realistic actor programs and specifications, the model checker can both detect protocol-violating bugs and prove conformance in a reasonable amount of time.

*A (Co)algebraic Approach to
Programming and Verifying Computer Networks*

STEFFEN JUILF SMOLKA
Cornell University, USA

Date: December 2019; Advisor: Nate Foster
URL: <https://tinyurl.com/sut55qr>

As computer networks have grown into some of the most complex and critical computing systems today, the means of configuring them have not kept up: they remain manual, low-level, and ad-hoc. This makes network operations expensive and network outages due to misconfigurations commonplace. The thesis of this dissertation is that high-level programming languages and formal methods can make network configuration dramatically easier and more reliable.

The dissertation consists of three parts. In the first part, we develop algorithms for compiling a network programming language with high-level abstractions to low-level network configurations, and introduce a symbolic data structure that makes compilation efficient in practice. In the second part, we develop foundations for a probabilistic network programming language using measure and domain theory, showing that continuity can be exploited to approximate (statistics of) packet distributions algorithmically. Based on this foundation and the theory of Markov chains, we then design a network verification tool that can reason about fault-tolerance and other probabilistic properties, scaling to data-center-size networks. In the third part, we introduce a general-purpose (co)algebraic framework for designing and reasoning about programming languages, and show that it permits an almost linear-time decision procedure for program equivalence. We hope that the framework will serve as a foundation for efficient verification tools, for networks and beyond, in the future.

Shared-Environment Call-by-Need

GEORGE WIDGERY STELLE
University of New Mexico, USA

Date: July 2019; Advisor: Darko Stefanovic
URL: <https://tinyurl.com/ujo2tzs>

Dissertation Abstract: Call-by-need semantics formalize the wisdom that work should be done at most once. It frees programmers to focus more on the correctness of their code, and less on the operational details. Because of this property, programmers of lazy functional languages rely heavily on their compiler to both preserve correctness and generate high-performance code for high level abstractions. In this dissertation I present a novel technique for compiling call-by-need semantics by using shared environments to share results of computation. I show how the approach enables a compiler that generates high-performance code, while staying simple enough to lend itself to formal reasoning. The dissertation is divided into three main contributions. First, I present an abstract machine, the CE machine, which formalizes the approach. Second, I show that it can be implemented as a native code compiler with encouraging performance results. Finally, I present a verified compiler, implemented in the Coq proof assistant, demonstrating how the simplicity of the approach enables formal verification.

Improving Haskell Transactional Memory

RYAN YATES

University of Rochester, USA

Date: October 2019; Advisor: Michael L. Scott

URL: <https://tinyurl.com/tx66gfe>

The Haskell programming language is an active laboratory for cutting edge ideas. Early in the evolution of transactional memory (TM), Haskell included language support and quickly grew a community of TM users. Since TM's inclusion in Haskell, a flurry of research has brought significant developments in TM in non-Haskell contexts including improved understanding of TM semantics, higher-performance implementations, and support for TM in commodity hardware. The community of Haskell TM users has continued to grow, largely due to composition and blocking features that are included in Haskell's TM but are typically missing from TM implementations in other languages. In this thesis we connect Haskell with new developments from the TM research community while preserving Haskell's rich TM features. We explore the challenges of integrating new ideas, including Transactional Locking II and hardware TM (HTM) into a pure functional programming language, and evaluate the performance of our developments. Achieving good cache performance, particularly avoiding speculative overflow, is critical to realizing benefits from HTM in its current form. To this end we implement a TM that removes indirection and allows for multiple transactional fields in a single heap object. To enable access to these features, we extend the Haskell language, implementing support for mutable constructor fields. Together these changes yield an implementation with significantly better performance than the original TM. We also explore the potential of using Haskell's advanced type system to decrease the cost of unused features. We argue that this can be achieved while still maintaining the existing API. More static information can be used both by the compiler to improve code and by users to better understand the performance characteristics of their code.
