JFP **27**, e25, 18 pages, 2017. © Cambridge University Press 2017
doi:10.1017/S0956796817000181

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, 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 12 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

Formal Dependability Analysis using Higher-Order-Logic Theorem Proving

WAQAR AHMAD
National University of Sciences and Technology, Pakistan

Date: August 2017; Advisor: Osman Hasan
URL: `http://tinyurl.com/ycjdwloe`

Dependability is an umbrella concept that subsumes many key properties about a system, including reliability, maintainability, safety, availability, confidentiality, and integrity. Various dependability modeling techniques have been developed to effectively capture the failure characteristics of systems over time. Traditionally, dependability models are analyzed using paper-and-pencil proof methods and computer based simulation tools but their results cannot be trusted due to their inherent inaccuracy limitations. To overcome these limitations, we propose to leverage upon the recent developments in probabilistic analysis support in higher-order-logic theorem proving to conduct accurate and rigorous dependability analysis. This thesis provides a semantic language embedding of the dependability concept that relies on a theory for probabilistic reasoning to develop a framework for formal dependability analysis within the sound environment of higher-order-logic theorem proving.

In this thesis, we mainly focus on the formalization of two widely used dependability modeling techniques: (i) Reliability Block Diagrams – a graphical technique used to determine the reliability of overall system by utilizing the failure characteristics of individual system components; and (ii) Fault Trees – used for graphically analyzing the conditions and the factors causing an undesired top event, i.e., a critical event, which can cause the whole system failure upon its occurrence. In particular, we present a RBD and FT-based formal dependability analysis framework that has the ability to accurately and rigorously determine the formal reliability, failure, availability and unavailability of safety-critical systems with arbitrary number of components. To illustrate the practical effectiveness of our proposed infrastructure, we present the formal dependability analysis of several real-world safety-critical systems, including smart grids, WSN data transport protocols, satellite solar arrays, virtual data centers, oil and gas pipeline systems and an air traffic management system using the HOL4 theorem prover.

Fibred Computational Effects

DANEL AHMAN
University of Edinburgh, UK

Date: September 2017; Advisor: Gordon Plotkin
URL: http://tinyurl.com/y9dofhoy

We study the interplay between *dependent types* and *computational effects*, two important areas of modern programming language research. On the one hand, dependent types underlie proof assistants such as Coq and functional programming languages such as Agda, Idris, and F*, providing programmers a means for encoding detailed specifications of program behaviour using types. On the other hand, computational effects, such as exceptions, nondeterminism, state, I/O, probability, etc., are integral to all widely-used programming languages, ranging from imperative languages, such as C, to functional languages, such as ML and Haskell. Separately, dependent types and computational effects both come with rigorous mathematical foundations, dependent types in the effect-free setting and computational effects in the simply typed setting. Their *combination*, however, has received much less attention and no similarly exhaustive theory has been developed. In this thesis we address this shortcoming by providing a comprehensive treatment of the combination of these two fields, and demonstrating that they admit a mathematically elegant and natural combination.

Specifically, we develop a core effectful dependently typed language, eMLTT, based on Martin-Löf's intensional type theory and a clear separation between (effect-free) values and (possibly effectful) computations familiar from simply typed languages such as Levy's Call-By-Push-Value and Egger et al.'s Enriched Effect Calculus. A novel feature of our language is the *computational sigma-type*, which we use to give a uniform treatment of type-dependency in sequential composition. In addition, we define and study a class of category-theoretic models, called *fibred adjunction models*, that are suitable for defining a sound and complete interpretation of eMLTT. Specifically, fibred adjunction models naturally combine standard category-theoretic models of dependent types (split closed comprehension categories) with those of computational effects (adjunctions). We discuss and study various examples of these models, including a domain-theoretic model, so as to extend eMLTT with general recursion.

We also investigate a dependently typed generalisation of the algebraic treatment of computational effects by showing how to extend eMLTT with *fibred algebraic effects* and their *handlers*. In particular, we specify fibred algebraic effects using a dependently typed generalisation of Plotkin and Pretnar's effect theories, enabling us to capture precise notions of computation such as state with location-dependent store types and dependently typed update monads. For handlers, we observe that their conventional term-level definition leads to unsound program equivalences becoming derivable in languages that include a notion of homomorphism, such as eMLTT.

To solve this problem, we propose a novel type-based treatment of handlers via a new computation type, the *user-defined algebra type*, which pairs a value type (the carrier) with a family of value terms (the operations). This type internalises Plotkin and Pretnar's insight that handlers denote algebras for a given equational theory of computational effects. We demonstrate the generality of our type-based treatment of handlers by showing that their conventional term-level presentation can be routinely derived, and that this treatment provides a useful mechanism for reasoning about effectful computations. Finally, we show that these extensions of eMLTT can be soundly interpreted in a fibred adjunction model based on the families of sets fibration and models of Lawvere theories.

Models of Type Theory with Strict Equality

PAOLO CAPRIOTTI
University of Nottingham, UK

Date: July 2016; Advisor: Venanzio Capretta
URL: `http://tinyurl.com/y9s7fwzq`

This thesis introduces the idea of *two-level type theory*, an extension of Martin Löf type theory that adds a notion of *strict equality* as an internal primitive.

A type theory with a strict equality alongside the more conventional form of equality, the latter being of fundamental importance for the recent innovation of *homotopy type theory* (HoTT), was first proposed by Voevodsky, and is usually referred to as HTS.

Here, we generalise and expand this idea, by developing a semantic framework that gives a systematic account of type formers for two-level systems, and proving a conservativity result relating back to a conventional type theory like HoTT.

Finally, we show how a two-level theory can be used to provide partial solutions to open problems in HoTT. In particular, we use it to construct semi-simplicial types, and lay out the foundations of an internal theory of $(\infty, 1)$-categories.

Dependent Pattern Matching and Proof-Relevant Unification

JESPER COCKX
Catholic University of Leuven, Belgium

Date: June 2017; Advisor: Frank Piessens and Dominique Devriese
URL: `http://tinyurl.com/yckcbovh`

Dependent type theory is a powerful language for writing functional programs with very precise types. It is used to write not only programs but also mathematical proofs that these programs satisfy certain properties. Because of this, languages based on dependent types – such as Coq, Agda, and Idris – are used both as programming languages and as interactive proof assistants. While dependent types give strong guarantees about your programs and proofs, they also impose equally strong requirements on them. This often makes it harder to write programs in a dependently typed language compared to one with a simpler type system. For this reason certain techniques have been developed, such as dependent pattern matching and specialization by unification. These techniques provide an intuitive way to write programs and proofs in dependently typed languages.

Previously, dependent pattern matching had only been shown to work in a limited setting. In particular, it relied on the K axiom – also known as the uniqueness of identity proofs – to remove equations of the form $x = x$. This axiom is inadmissible in many type theories, particularly in the new and promising branch known as homotopy type theory (HoTT). As a result, programs and proofs in these new theories cannot make use of dependent pattern matching and are as a result much harder to write, modify, and understand. Additionally, the interaction of dependent pattern matching with small but practical features such as eta-equality for record types and postponing of unification constraints was poorly understood, resulting in subtle bugs and inconsistencies. In this thesis, we develop dependent pattern matching and unification in a general setting that does not require the K axiom, both from a theoretical perspective and a practical one. In particular, we present a proof-relevant unification algorithm, where each unification rule produces evidence of its correctness. This evidence guarantees that all unification rules are correct by construction, and also gives a computational characterization to each unification rule.

To ensure that these techniques are sound and will stay so in face of future extensions to type theory, we show how to translate them to more basic primitive constructs, i.e. the standard datatype eliminators. During this translation, we pay special attention to the computational content of all constructions involved. This guarantees that the intuitions from regular pattern matching carry over to a dependently typed setting.

Based on our work, we implemented a complete overhaul of the algorithm for checking definitions by dependent pattern matching in Agda. Our new implementation fixes a substantial number of issues in the old implementation, and is at the same time less restrictive than the old ad-hoc restrictions. Thus it puts the whole system back on a strong foundation. In addition, our work has already been used as the basis for other implementations of dependent pattern matching, such as the Equations package for Coq and the Lean theorem prover. The work in this thesis eliminates all implicit assumptions introduced to the type theory by pattern matching and unification. In the future, we may also want to integrate new principles with pattern matching, for example the higher inductive types introduced by HoTT. The framework presented in this thesis also provides a solid basis for such extensions to be built on.

Mechanizing Abstract Interpretation

DAVID DARAIS
University of Maryland, USA

Date: August 2017; Advisor: David Van Horn
URL: `http://tinyurl.com/y8pnr8jp`

It is important when developing software to verify the absence of undesirable behavior such as crashes, bugs and security vulnerabilities. Some settings require high assurance in verification results, *e.g.*, for embedded software in automobiles or airplanes. To achieve high assurance in these verification results, formal methods are used to automatically construct or check proofs of their correctness. However, achieving high assurance for program analysis results is challenging, and current methods are ill suited for both complex critical domains and mainstream use.

To verify the correctness of software we consider *program analyzers*—automated tools which detect software defects—and to achieve high assurance in verification results we consider *mechanized verification*—a rigorous process for establishing the correctness of program analyzers via computer-checked proofs.

The key challenges to designing verified program analyzers are: (1) achieving an analyzer *design* for a given programming language and correctness property; (2) achieving an *implementation* for the design; and (3) achieving a *mechanized verification* that the implementation is correct w.r.t. the design. The state of the art in (1) and (2) is to use *abstract interpretation*: a guiding mathematical framework for systematically constructing analyzers directly from programming language semantics. However, achieving (3) in the presence of abstract interpretation has remained an open problem since the late 1990's. Furthermore, even the state-of-the art which achieves (3) in the absence of abstract interpretation suffers from the inability to be reused in the presence of new analyzer designs or programming language features.

First, we solve the open problem which has prevented the combination of abstract interpretation (and in particular, *calculational* abstract interpretation) with mechanized verification, which advances the state of the art in designing, implementing, and verifying analyzers for critical software. We do this through a new mathematical framework *Constructive Galois Connections* which supports synthesizing specifications for program analyzers, calculating implementations from these induced specifications, and is amenable to mechanized verification.

Finally, we introduce reusable components for implementing analyzers for a wide range of designs and semantics. We do this though two new frameworks *Galois Transformers* and *Definitional Abstract Interpreters*. These frameworks tightly couple analyzer design decisions, implementation fragments, and verification properties into compositional components which are (target) programming-language independent and amenable to mechanized verification. Variations in the analysis design are then recovered by simply re-assembling the combination of components. Using this framework, sophisticated program analyzers can be assembled by non-experts, and the result are guaranteed to be verified by construction.

Fully Generic Programming Over Closed Universes of Inductive-Recursive Types

LARRY DIEHL
Portland State University, USA

Date: June 2017; Advisor: Tim Sheard
URL: `http://tinyurl.com/y8e4hbgy`

Dependently typed programming languages allow the type system to express arbitrary propositions of intuitionistic logic, thanks to the Curry-Howard isomorphism. Taking full advantage of this type system requires defining more types than usual, in order to encode logical correctness criteria into the definitions of datatypes. While an abundance of specialized types helps ensure correctness, it comes at the cost of needing to redefine common functions for each specialized type.

This dissertation makes an effort to attack the problem of code reuse in dependently typed languages. Our solution is to write generic functions, which can be applied to any datatype. Such a generic function can be applied to datatypes that are defined at the time the generic function was written, but they can also be applied to any datatype that is defined in the *future*. Our solution builds upon previous work on generic programming within dependently typed programming.

Type theory supports generic programming using a construction known as a *universe*. A universe can be considered the model of a programming language, such that writing functions over it models writing generic programs in the programming language. Historically, there has been a trade-off between the expressive power of the modeled programming language, and the kinds of generic functions that can be written in it. Our dissertation shows that no such trade-off is necessary, and that we can write future-proof generic functions in a model of a dependently typed programming language with a rich collection of types.

Sequent Calculus: A Logic and a Language for Computation and Duality

PAUL DOWNEN
University of Oregon, USA

Date: June 2017; Advisor: Zena M. Ariola
URL: `http://tinyurl.com/yb8fy94n`

Truth and falsehood, questions and answers, construction and deconstruction; most things come in dual pairs. Duality is a mirror that reveals the new from the old via opposition. This idea appears pervasively in logic, where duality inverts "true" with "false" and "and" with "or." However, even though programming languages are closely connected to logics, this kind of strong duality is not so apparent in practice. Sum types (disjoint tagged unions) and product types (structures) are dual concepts, but in the realm of programming, natural biases obscure their duality.

To better understand the role of duality in programming, we shift our perspective. Our approach is based on the *Curry-Howard isomorphism* which says that programs following a specification are the same as proofs for mathematical theorems. This thesis explores Gentzen's sequent calculus, a logic steeped in duality, as a model for computational duality. By applying the Curry-Howard isomorphism to the sequent calculus, we get a language that combines dual programming concepts as equal opposites: data types found in functional languages are dual to co-data types (interface-based objects) found in object-oriented languages, control flow is dual to information flow, induction is dual to co-induction. This gives a duality-based semantics for reasoning about programs via *orthogonality*: checking safety and correctness based on a comprehensive test suite.

We use the language of the sequent calculus to apply ideas from logic to issues relevant to program compilation. The idea of *logical polarity* reveals a symmetric basis of primitive programming constructs that can faithfully represent all user-defined data and co-data types. We reflect the lessons learned back into a core language for functional languages, at the cost of symmetry, with the relationship between the sequent calculus and natural deduction. This relationship lets us derive a pure λ-calculus with user-defined data and co-data which we further extend by bringing out the implicit control-flow in functional programs. Explicit control-flow lets us share and name control the same way we share and name data, enabling a direct representation of *join points*, which are essential for tractable optimization and compilation.

Higher-Dimensional Types in the Mechanization of Homotopy Theory

KUEN-BANG HOU (FAVONIA)
Carnegie Mellon University, USA

Date: May 2017; Advisor: Robert Harper
URL: `http://tinyurl.com/ydas4saa`

Mechanized reasoning has proved effective in avoiding serious mistakes in software and hardware, and yet remains unpopular in the practice of mathematics. My thesis is aimed at making mechanization easier so that more mathematicians can benefit from this technology. Particularly, I experimented with higher-dimensional types, an extension of ordinary types with a hierarchy of stacked relations, and managed to mechanize many important results from classical homotopy theory in the proof assistant Agda. My work thus suggests higher-dimensional types may help mechanize mathematical concepts.

Web Session Security: Formal Verification, Client-Side Enforcement and Experimental Analysis

WILAYAT KHAN
Ca' Foscari University of Venice, Italy

Date: February 2015; Advisor: Michele Bugliesi
URL: `http://tinyurl.com/y8fds8bk`

Web applications are the dominant means to provide access to millions of on-line services and applications such as banking and e-commerce. To personalize users web experience, servers need to authenticate the users and then maintain their authentication state throughout a set of related HTTP requests and responses called a *web session*. As HTTP is a stateless protocol, the common approach, used by most of the web applications to maintain web session, is to use HTTP cookies. Each request belonging to a web session is authenticated by having the web browser to provide to the server a unique long random string, known as *session identifier* stored as cookie called *session cookie*. Taking over the session identifier gives full control over to the attacker and hence is an attractive target of the attacker to attack on the confidentiality and integrity of web sessions. The browser should take care of the *web session security*: a session cookie belonging to one source should not be corrupted or stolen or forced, to be sent with the requests, by any other source.

This dissertation demonstrates that security policies can in fact be written down for both, confidentiality and integrity, of web sessions and enforced at the client side without getting any support from the servers and without breaking too many web applications. Moreover, the enforcement mechanisms designed can be proved correct within mathematical models of the web browsers. These claims are supported in this dissertation by 1) defining both, end-to-end and access control,security policies to protect web sessions; 2) introducing a new and using exiting mathematical models of the web browser extended with confidentiality and integrity security policies for web sessions; 3) offering mathematical proofs that the security mechanisms do enforce the security policies; and 4) designing and developing prototype browser extensions to test that real-life web applications are supported.

Streaming for Functional Data-Parallel Languages

FREDERIK MEISNER MADSEN
University of Copenhagen, Denmark

Date: September 2016; Advisor: Andrzej Filinski
URL: `http://tinyurl.com/yddxykgw`

In this thesis, we investigate streaming as a general solution to the space inefficiency commonly found in functional data-parallel programming languages. The data-parallel paradigm maps well to parallel SIMD-style hardware. However, the traditional fully materializing execution strategy, and the limited memory in these architectures, severely constrains the data sets that can be processed. Moreover, the language-integrated cost semantics for nested data parallelism pioneered by NESL depends on a parallelism- flattening execution strategy that only exacerbates the problem. This is because flattening necessitates all sub-computations to materialize at the same time. For example, naive n by n matrix multiplication requires n^3 space in NESL because the algorithm contains n^3 independent scalar multiplications. For large values of n, this is completely unacceptable.

We address the problem by extending two existing data-parallel languages: NESL and Accelerate. In the extensions we map bulk operations to data-parallel streams that can evaluate fully sequential, fully parallel or anything in between. By a dataflow, piecewise parallel execution strategy, the runtime system can adjust to any target machine without any changes in the specification. We expose streams as sequences in the frontend languages to provide the programmer with high-level information and control over streamable and non-streamable computations. In particular, we can extend NESLs intuitive and high-level workdepth model for time complexity with similarly intuitive and high-level model for space complexity that guarantees streamability.

Our implementations are backed by empirical evidence. For Streaming Accelerate we demonstrate performance on par with Accelerate without streams for a series of benchmark including the PageRank algorithm and a MD5 dictionary attack algorithm. For Streaming NESL we show that for several examples of simple, but not trivially parallelizable, text-processing tasks, we obtain single-core performance on par with off-the-shelf GNU Coreutils code, and near-linear speedups for multiple cores.

Reasonably Programmable Syntax

CYRUS OMAR
Carnegie Mellon University, USA

Date: May 2017; Advisor: Jonathan Aldrich
URL: `http://tinyurl.com/y7aoqcrt`

Programming languages commonly provide "syntactic sugar" that decreases the syntactic cost of working with certain standard library constructs. For example, Standard ML builds in syntactic sugar for constructing and pattern matching on lists. Third-party library providers are, justifiably, envious of this special arrangement. After all, it is not difficult to find other situations where library-specific syntactic sugar might be useful. For example, (1) clients of a "collections" library might like syntactic sugar for finite sets and dictionaries; (2) clients of a "web programming" library might like syntactic sugar for HTML and JSON values; (3) a compiler writer might like syntactic sugar for the terms of the object language or various intermediate languages of interest; and (4) clients of a chemistry library might like syntactic sugar for chemical structures based on the SMILES standard.

Defining a "library-specific" syntax dialect in each of these situations is problematic, because library clients cannot combine dialects like these in a manner that conserves syntactic determinism (i.e. syntactic conflicts can and do arise.) Moreover, it can become difficult for library clients to reason abstractly about types and binding when examining the text of a program that uses unfamiliar forms. Instead, they must reason transparently, about the underlying expansion. Typed, hygienic term-rewriting macro systems, like Scalas macro system, while somewhat more reasonable, offer limited syntactic control.

This thesis formally introduces *typed literal macros (TLMs)*, which give library providers the ability to programmatically control the parsing and expansion, at "compile-time", of expressions and patterns of *generalized literal form*. Library clients can use any combination of TLMs in a program without needing to consider the possibility of syntactic conflicts between them, because the context-free syntax of the language is never extended (instead, literal forms are contextually repurposed.) Moreover, the language validates each expansion that a TLM generates in order to maintain useful abstract reasoning principles. Most notably, expansion validation maintains:

- a *type discipline*, meaning that the client can reason about types while holding the literal expansion abstract; and
- a *strictly hygienic binding discipline*, meaning that the client can always be sure that:
  1. spliced terms, i.e. terms that appear within literal bodies, cannot capture bindings hidden within the literal expansion; and

2. the literal expansion does not refer to definition-site or applicationsite bindings directly. Instead, all interactions with bindings external to the expansion go explicitly through spliced terms or parameters.

In short, we formally define a programming language in the ML tradition with a *reasonably* programmable syntax.

A Type-Theoretical Study of Non-Termination

NICCOLÒ VELTRI
Tallinn University of Technology, Estonia

Date: May 2017; Advisor: Tarmo Uustalu and James Chapman
URL: `http://tinyurl.com/y82dmtk9`

In this thesis, we continue the study of Capretta's coinductive delay monad in Martin-Löf type theory. The delay monad constitutes a viable constructive alternative to the maybe monad and allows the implementation of possibly non-terminating computations. Its applications range from the representation of general recursive functions to the formalization of domain theory, from the operational semantics for While languages to normalization by evaluation.

In all these applications, one is only interested in the terminating/non-terminating behavior of a computation, and not in its rate of convergence. This is equivalent to working with the delay datatype quotiented by weak bisimilarity. Our first main result is the discovery that the delay datatype quotiented by weak bisimilarity does not inherit the monad structure immediately. This has to do with the coinductive nature of the delay datatype and the bad interaction between inductive-like quotients in the style of Hofmann and infinitary types such as non-wellfounded trees. In order to construct a monad structure on the delay type, we need to postulate additional classical or semi-classical principles, such as the limited principle of omniscience or a certain weak version of the axiom of countable choice. These principles are also necessary for proving that the quotiented delay monad delivers free ω-complete pointed partial orders.

We can say that the quotiented delay monad is an useful tool for modeling partiality as an effect in type theory. Our second main result is to make the latter statement rigorous. We introduce a class of monads for encoding non-termination as an effect. A monad in this class is named a ω-complete pointed classifying monad, which formally is a monad whose Kleisli category is a restriction category à la Cockett and Lack, which moreover is ωCPPO-enriched with respect to the restriction order and in which pure maps are total. We show that the quotiented delay monad is the initial ω-complete pointed classifying monad in type theory. This universal property singles it out from among other examples of such monads, for examples from other partial map classifiers specified in terms of countably-complete dominances.

From a more general point of view, we ask ourselves whether type-theoretical approaches to partiality could possibly benefit from category-theoretical ones. Although we do not have a complete answer to this question yet, we present the first steps in this direction. Our last main result consists of an Agda formalization of the first chapters of the theory of Cockett and Lack's restriction categories. Notably, it includes the proof of their completeness with respect to partial map categories, the latter being the standard generalization of sets and partial functions to more general categories. We hope that our development can become the cornerstone of a flexible framework for partiality in dependently typed programming languages, allowing one to program and reason about partial functions on different levels of abstraction.