JFP **29**, e10, 14 pages, 2019.
doi:10.1017/S0956796819000078

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 thirteen 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

Data-Driven Refactorings for Haskell

STEPHEN ADAMS
University of Kent, UK

Date: August 2017; Advisor: Simon Thompson
URL: https://tinyurl.com/y236dprt

Agile software development allows for software to evolve slowly over time. Decisions made during the early stages of a program's lifecycle often come with a cost in the form of technical debt. Technical debt is the concept that reworking a program that is implemented in a naive or "easy" way, is often more difficult than changing the behaviour of a more robust solution. Refactoring is one of the primary ways to reduce technical debt. Refactoring is the process of changing the internal structure of a program without changing its external behaviour. The goal of performing refactorings is to increase code quality, maintainability, and extensibility of the source program. Performing refactorings manually is time consuming and error-prone. This makes automated refactoring tools very useful. Haskell is a strongly typed, pure functional programming language. Haskell's rich type system allows for complex and powerful data models and abstractions. These abstractions and data models are an important part of Haskell programs. This thesis argues that these parts of a program accrue technical debt, and that refactoring is an important technique to reduce this type of technical debt. Refactorings exist that tackle issues with a program's data model, however these refactorings are specific to the object-oriented programming paradigm. This thesis reports on work done to design and automate refactorings that help Haskell programmers develop and evolve these abstractions. This work also discussed the current design and implementation of HaRe (the *Ha*skell *Re*factorer). HaRe now supports the Glasgow Haskell Compiler's implementation of the Haskell 2010 standard and its extensions, and uses some of GHC's internal packages in its implementation.

Quantitative Aspects and Generation of Random Lambda and Combinatory Logic Terms

MACIEJ BENDKOWSKI
Jagiellonian University, Poland

Date: November 2017; Advisor: Marek Zaionc and Katarzyna Grygiel
URL: https://tinyurl.com/y4oyo44c

We present a quantitative analysis of lambda-calculus in the de Bruijn notation and combinatory logic under various combinator bases. Both classes of computational models are shown to share the fixed subterm property – for an arbitrary fixed term T, asymptotically almost all terms contain T as a subterm. In consequence, both models exhibit similar quantitative properties with respect to normalisation and typeability. Specifically, asymptotically almost no term is either strongly normalising or typeable. Furthermore, asymptotically almost no normalising term is simultaneously strongly normalising.

Concerning combinator-specific results, we provide a complete syntactic characterisation of normalising SK-combinators by means of a constructive hierarchy of unambiguous regular tree grammars. As an application, we present an algorithmic technique of finding asymptotically significant fractions of normalising SK-combinators. Utilising our systematic approach, we show that the asymptotic density of normalising combinators cannot be less than 34%. We discuss the limits of our method and, based on super-computer experimental results, discuss the asymptotic density and average complexity of normalising combinators, arguing that the asymptotic density of normalising combinators is approximately equal to 85%.

Finally, we discuss the effective generation of random lambda-terms and combinators, focusing on the set of closed and typeable lambda-terms. We provide effective Boltzmann samplers for several classes of interesting lambda-terms including the restricted class of so-called closed h-shallow lambda-terms, i.e. closed lambda-terms in which de Bruijn indices are bounded by h. Combining Boltzmann models and logic programming techniques available in modern Prolog systems, we give a sampling scheme for closed typeable lambda-terms and discuss the intriguing challenges blocking effective sampling of large closed typeable lambda-terms.

https://doi.org/10.1017/S0956796819000078 Published online by Cambridge University Press

Disjoint Intersection Types: Theory and Practice

XUAN BI
University of Hong Kong

Date: December 2018; Advisor: Bruno Oliveira
URL: https://tinyurl.com/y4foktdw

Programs are hard to write. It was so 50 years ago at the time of the so-called software crisis; it still remains so nowadays. Over the years, we have learned—the hard way—that software should be constructed in a modular way, i.e., as a network of smaller and loosely connected modules. To facilitate writing modular code, researchers and software practitioners have developed new methodologies; new programming paradigms; stronger type systems; as well as better tooling support. Still, this is not enough to cope with today's needs. Several reasons have been raised for the lack of satisfactory solutions, but one that is constantly pointed out is the inadequacy of existing programming languages for the construction of modular software. This thesis investigates disjoint intersection types, a variant of intersection types. Disjoint intersections types have great potential to serve as a foundation for powerful, flexible and yet type-safe and easy to reason OO languages, suitable for writing modular software. On the theoretical side, this thesis shows how to significantly increase the expressiveness of disjoint intersection types by adding support for nested composition, along with parametric polymorphism. Nested composition extends inheritance to work on a whole family of classes, enabling high degrees of modularity and code reuse. The combination with parametric polymorphism further improves the state-of-art encodings of extensible designs. However, the extension with nested composition and parametric polymorphism is challenging, for two different reasons. Firstly, the subtyping relation that supports these features is non-trivial. Secondly, the syntactic method used to prove coherence for previous calculi with disjoint intersection types is too inflexible. This thesis addresses the first problem by adapting and extending the well-known BCD subtyping with records, universal quantification and coercions. To address the second problem, this thesis proposes a powerful proof method to establish coherence. Hence, this thesis puts disjoint intersection types on a solid footing by thoroughly exploring their meta-theoretical properties.

Compiling with Dependent Types

WILLIAM J. BOWMAN
Northeastern University, USA

Date: May 2019; Advisor: Amal Ahmed
URL: https://tinyurl.com/y3d9s5zt

Dependently typed languages have proven useful for developing large-scale fully verified software, but we do not have any guarantees after compiling that verified software. A verified program written in a dependently typed language, such as Coq, can be type checked to ensure that the program meets its specification. Similarly, type checking prevents us from importing a library and violating the specification declared by its types. Unfortunately, we cannot perform either of these checks after compiling a dependently typed program, since all current implementations erase types before compiling the program. Instead, we must trust the compiler to not introduce errors into the verified code, and, after compilation, trust the programmer to never introduce errors by linking two incompatible program components. As a result, the compiled and linked program is not verified—we have no guarantees about what it will do.

In this dissertation, I develop a theory for preserving dependent types through compilation so that we can use type checking after compilation to check that no errors are introduced by the compiler or by linking. Type-preserving compilation is a well-known technique that has been used to design compilers for non-dependently typed languages, such as ML, that statically enforce safety and security guarantees in compiled code. But there are many open challenges in scaling type preservation to dependent types. The key problems are adapting syntactic type systems to interpret low-level representations of code, and breaking the complex mutually recursive structure of dependent type systems to make proving type preservation and compiler correctness feasible. In this dissertation, I explain the concepts required to scale type preservation to dependent types, present a proof architecture and language design that support type preservation, and prove type preservation and compiler correctness for four early-stage compiler translations of a realistic dependently typed calculus. These translations include an A-normal form (ANF), a continuation-passing style (CPS), an abstract closure conversion, and a parametric closure conversion translation.

Abstracting Control with Dependent Types

YOUYOU CONG
Ochanomizu University, Japan

Date: March 2019; Advisor: Kenichi Asai
URL: https://tinyurl.com/yyegepx3

Dependent types are a powerful tool for ensuring safety. By interacting with terms, dependent types are able to precisely encode program specifications, guaranteeing the absence of runtime errors and other unexpected behaviours. Meanwhile, control operators have been extensively used to increase expressiveness. By talking about the surroundings of programs, control operators enable sophisticated manipulation of control flow, yielding a wide range of practical applications.

The two language ingredients are however known to pose various difficulties when mixed up together. Intuitively, the disharmony stems from their contrasting nature: dependent types are used for reasoning purposes and thus must be determined statically, whereas control operators are used to implement dynamic, non-local behaviours. To make their combination meaningful, previous work has imposed a purity restriction on type dependency, that is, types may depend only on effect-free terms.

In this thesis, we build a dependently typed, effectful language called Dellina. Dellina has support for essential features from the mainstream proof assistants, as well as the delimited control operators shift and reset. Similarly to existing studies, we restrict types to depend only on pure terms, but additionally, we impose two constraints on the type of contexts surrounding effectful terms, in order to cope with the flexibility of the control operators. These restrictions make the resulting language type sound. We also define a selective CPS translation of the language, and prove that the translation preserves typing. Our key observation is that, in a dependently typed setting, selective translations not only yield efficient programs, but simplify the proof of the type preservation property.

Dellina is the first non-toy language where dependent types and control operators coexist. As an application, we show an implementation of a type-safe evaluator that uses shift, reset, and dependent types all in a non-trivial manner. Our result further opens the door to integrating shift and reset into proof assistants. We discuss how we should extend the "proofs-as-programs" view to a language with delimited control, and what we can prove with the control operators.

Model Construction, Evolution, and Use in Testing of Software Systems

PABLO LAMELA SEIJAS
University of Kent, UK

Date: July 2018; Advisor: Simon Thompson
URL: https://tinyurl.com/y6hquyaf

The ubiquity of software places emphasis on the need for techniques that allow us to ensure that software behaves as we expect it to behave. The most widely-used approach to ensuring software quality is unit testing, but this is arguably not a very efficient solution, since each test only checks that the software behaves as expected in one single scenario.

There exist more advanced techniques, like property-based testing, model-checking, and formal verification, but they usually rely on properties, models, and specifications. One source of friction faced by testers that want to use these advanced techniques is that they require the use of abstraction and, as humans, we tend to find it more difficult to think of abstract specifications than to think of concrete examples.

In this thesis, we study how to make it easier to create models that can be used for testing software. In particular, we research the creation of reusable models, ways of automating the generalisation of code and models, and ways of automating the generation of models from legacy unit tests and execution traces.

As a result, we provide techniques for generating tests from state machine models, techniques for inferring parametrised state machines from code, and refactorings that automate the introduction of abstraction for property-based testing models and code in general. All these techniques are illustrated with concrete examples and with open-source implementations that are publicly available.

Tools and Techniques for the Verification of Modular Stateful Code

MÁRIO JOSÉ PARREIRA PEREIRA
Université Paris-Saclay, France

Date: December 2018; Advisor: Jean-Christophe Filliâtre
URL: https://tinyurl.com/y3jc4xr8

This thesis is set in the field of formal methods, more precisely in the domain of deductive program verification. Throughout this thesis, the Why3 verification framework is used, on one hand, as a working environment to experiment and validate our research and, on the other hand, as the direct target of certain contributions of this thesis. Why3 features an ML-like language called WhyML, which is both a programming and a specification language. An important aspect of WhyML is the presence of *ghost code*, i.e. programming elements that are introduced exclusively for specification and proof purposes. In order to get an executable code, ghost code must be eliminated via an automated translation called *extraction*. One of the main contributions of this thesis is the formalization and implementation of an extraction mechanism for Why3.

The new extraction mechanism for Why3 is successfully used to generate several correct-by-construction OCaml modules, in the scope of a larger research project aiming at building a formally verified general-purpose library of efficient data structures and algorithms. This thesis also contributes to the development of a formal specification language for OCaml. We use this language to equip the library API with a specification, and we use our extraction mechanism to provide verified implementations.

While building this verified library, we were naturally confronted with the proof of idiomatic OCaml features which are beyond the scope of Why3. This led to three other contributions of this thesis. The first is a systematic technique for the verification of pointer-based data structures, building on explicit memory models and local reasoning on delimited portions of the heap. Using that technique, we were able to provide a fully automated proof of a union-find implementation. The second is an approach to extract functorial code from verified WhyML programs. This allows us to extract OCaml functors for different implementations of parameterized data structures. Last, we propose a general and modular way to specify programs performing iteration, independently of the underlying implementation paradigm. Several cursors and higher-order iterators have been specified and verified using such an approach.

Polymorphic Set-Theoretic Types for Functional Languages

TOMMASO PETRUCCIANI
Università di Genova, Italy and Université Sorbonne Paris Cité, France

Date: March 2019; Advisor: Giuseppe Castagna and Elena Zucca
URL: https://tinyurl.com/yxgjtxkl

We study *set-theoretic types*: types that include union, intersection, and negation connectives. Set-theoretic types, coupled with a suitable subtyping relation, are useful to type several programming language constructs including conditional branching, pattern matching, and function overloading very precisely. We define subtyping following the *semantic subtyping* approach, which interprets types as sets and defines subtyping as set inclusion. Our set-theoretic types are *polymorphic*, that is, they contain type variables to allow parametric polymorphism.

We extend previous work on set-theoretic types and semantic subtyping by showing how to adapt them to new settings and apply them to type various features of functional languages. More precisely, we integrate semantic subtyping with three important language features.

In Part I we study implicitly typed languages with let-polymorphism and type inference (previous work on semantic subtyping focused on explicitly typed languages). We describe an implicitly typed lambda-calculus and a declarative type system for which we prove soundness. We study type inference and prove results of soundness and completeness. Then, we show how to make type inference more precise when programs are partially annotated with types.

In Part II we study gradual typing. We describe a new approach to add gradual typing to a static type system; the novelty is that we give a declarative presentation of the type system, while previous work considered algorithmic presentations. We first illustrate the approach on a Hindley-Milner type system without subtyping. We describe declarative typing, compilation to a cast language, and sound and complete type inference. Then, we add set-theoretic types, defining a subtyping relation on set-theoretic gradual types, and we describe sound type inference for the extended system.

In Part III we consider non-strict semantics. The existing semantic subtyping systems are designed for call-by-value languages and are unsound for non-strict semantics. We adapt them to obtain soundness for call-by-need. To do so, we introduce an explicit representation for divergence in the types, allowing the type system to distinguish the expressions that are already evaluated from those that are computations which might diverge.

Automatic Generation of Proof Terms in Dependently Typed Programming Languages

FRANCK SLAMA
University of St Andrews, UK

Date: September 2018; Advisor: Edwin Brady
URL: https://tinyurl.com/y29r8ns8

Dependent type theories are a kind of mathematical foundations investigated both for the formalisation of mathematics and for reasoning about programs. They are the theories of many proof assistants and programming languages with proofs (Coq, Agda, Idris, Dedukti, Matita, etc). Dependent types allow to encode elegantly and constructively the universal and existential quantifications of higher-order logics and are therefore adapted for writing logical propositions and proofs. However, their usage is not limited to the area of pure logic. Indeed, some recent work has shown that they can also be powerful for driving the construction of programs. Using more precise types not only helps to gain confidence about the program built, but it can also help its construction, giving rise to a new style of programming called Type-Driven Development. However, one difficulty with reasoning and programming with dependent types is that proof obligations arise naturally once programs become even moderately sized. The need for these proofs comes, in intensional type theories (like CIC and ML) from the fact that in a non-empty context, the propositional equality allows us to prove as equal (with the induction principles) terms that are not judgementally equal, which implies that the typechecker can't always obtain equality proofs by reduction. As far as possible, we would like to solve such proof obligations automatically, and we absolutely need it if we want dependent types to be used more broadly, and to become one day the standard in functional programming. In this thesis, we show one way to automate these proofs by reflection in the dependently typed programming language Idris. However, the method that we follow is language independent. We present an original type-safe reflection mechanism, where reflected terms are indexed by the original Idris expression that they represent, and show how it allows to easily construct and manipulate proofs. We build a hierarchy of correct-by-construction tactics for proving equivalences in semi-groups, monoids, commutative monoids, groups, commutative groups, semi-rings and rings. We also show how each tactic reuses those from simpler structures, thus avoiding duplication of code and proofs. Finally, and as a conclusion, we discuss the trust we can have in such machine-checked proofs.

Executable Formal Specification of Programming Languages with Reusable Components

L. THOMAS VAN BINSBERGEN
Royal Holloway, UK

Date: April 2019; Advisor: Adrian Johnstone and Elizabeth Scott
URL: https://tinyurl.com/y24xdyvm

Writing a formal definition of a programming language is cumbersome and maintaining the definition as the language evolves is tedious and error-prone. But programming languages have commonalities that can be captured once and for all and used in the formal definition of multiple languages, potentially easing the task of developing and maintaining definitions. Languages often share features, even across paradigms, such as scoping rules, function calls, and control operators. Moreover, some concrete syntaxes share patterns such as repetition with a separator, delimiters around blocks, and prefix and infix operators.

The PLanCompS project has established a formal and component-based approach to semantics intended to reduce development and maintenance costs by employing the software engineering practices of reuse and testing. This thesis contributes further, taking advantage of the advanced features of the Haskell programming language to define *executable* and *reusable* components for specifying both syntax and semantics.

The main theoretical contributions of this thesis are: a data structure for representing the possibly many derivations found by a generalised parser which significantly simplifies the specification of generalised parsing algorithms, a purely functional description of GLL parsing based on this data structure, a novel approach to combinator parsing that incorporates generalised parsing algorithms such as GLL in their implementation, and a novel and lightweight framework for developing modular rule-based semantic specifications (MRBS). This thesis shows how languages with Homogeneous Generative Meta-Programming facilities are formalised within the presented approach.

The main practical contributions of this thesis are: a combinator library for component-based descriptions of context-free grammars from which GLL parsers are generated, a compiler and interpreter for executing operational semantic specifications written in CBS (the meta-language developed by the PLanCompS project), an interpreter for an intermediate meta-language (IML) based on MRBS, and a translation from CBS specifications into IML that gives an operational semantics to CBS. A language definition written with these tools is executable so that a prototype implementation of the language can be generated from its definition.

The practicality of the presented tools is evaluated through case studies.

Verifying Information Flow Control Libraries

MARCO VASSENA
Chalmers University of Technology, Sweden

Date: February 2019; Advisor: Alejandro Russo
URL: https://tinyurl.com/y5steb99

Information Flow Control (IFC) is a principled approach to protecting the confidentiality and integrity of data in software systems. Intuitively, IFC systems associate data with security labels that track and restrict flows of information throughout a program in order to enforce security. Most IFC techniques require developers to use specific programming languages and tools that require substantial efforts to develop or to adopt. To avoid redundant work and lower the threshold for adopting secure languages, IFC has been embedded in general-purpose languages through software libraries that promote *security-by-construction* with their API.

This thesis makes several contributions to state-of-the-art static (MAC) and dynamic IFC libraries (LIO) in three areas: expressive power, theoretical IFC foundations and protection against covert channels. Firstly, the thesis gives a *functor algebraic* structure to sensitive data, in a way that it can be processed through classic functional programming patterns that do not incur in security checks. Then, it establishes the formal security guarantees of MAC, using the standard proof technique of term erasure, enriched with *two-steps erasure*, a novel idea that simplifies reasoning about advanced programming features, such as exceptions, mutable references and concurrency. Secondly, the thesis demonstrates that the lightweight, but coarse-grained, enforcement of dynamic IFC libraries (e.g., LIO) can be as precise and permissive as the fine-grained, but heavyweight, approach of fully-fledged IFC languages. Lastly, the thesis contributes to the design of secure runtime systems that protect IFC libraries, and IFC languages as well, against internal- and external-timing covert channels that leak information through certain runtime system resources and features, such as *lazy evaluation* and *parallelism*.

The results of this thesis are supported with extensive machine-checked proof scripts, consisting of 12,000 lines of code developed in the Agda proof assistant.

Revealing Behaviours of Concurrent Functional Programs by Systematic Testing

MICHAEL STEWART WALKER
University of York, UK

Date: March 2019; Advisor: Colin Runciman
URL: https://tinyurl.com/yxqgu9ow

We aim to make it easier for programmers to write correct concurrent programs and to demonstrate that concurrency testing techniques, typically described in the context of simple core languages, can be successfully applied to languages with more complex concurrency. In pursuit of these goals, we develop three lines of work:

Testing concurrent Haskell We develop a library for testing concurrent Haskell programs using a typeclass abstraction of concurrency, which we give a formal semantics. Our tool implements *systematic concurrency testing*, a family of techniques for deterministically testing concurrent programs. Along the way we also tackle how to soundly handle daemon threads, and how to usefully present complex execution traces to a user. We not only obtain a useful tool for Haskell programs, but we also show that these techniques work well in languages with rich concurrency abstractions.

Randomised concurrency testing We propose a new algorithm for *randomly* testing concurrent programs. This approach is fundamentally incomplete, but can be suitable in cases where systematic concurrency testing is not. We show that our algorithm performs as well as a pre-existing popular algorithm for a standard set of benchmarks. This pre-existing algorithm requires the use of program-specific parameters, but our algorithm does not. We argue that this makes use and implementation of our algorithm simpler.

Finding properties of programs We develop a tool for finding properties of sets of concurrency functions operating on some shared state, such as the API for a concurrent data type. Our tool enumerates Haskell expressions and discovers properties by comparing execution results for a variety of inputs. Unlike other property discovery tools, we support side effects. We do so by building on our tool for testing concurrent Haskell programs. We argue that this approach can lead to greater understanding of concurrency functions.

Safely Exposing Haskell's Hidden Powers

THOMAS WINANT
KU Leuven, Belgium

Date: November 2018; Advisor: Frank Piessens and Dominique Devriese
URL: https://tinyurl.com/y5ghr6oa

Haskell is a very influential functional programming language with an advanced type system that is capable of detecting many bugs. In this dissertation, we are interested in situations where the Glasgow Haskell Compiler (GHC), the de facto standard compiler of Haskell, is more powerful than the Haskell programming language itself: it has the capability to do things programmers cannot do using the language. Very often, these capabilities are not made available to programmers or only in restricted forms, because doing so would be unsafe, i.e. it would potentially break safety properties of Haskell or internal invariants of the compiler. Unfortunately, such compromises mean that useful functionality is unavailable to Haskell programmers for safety reasons. We refer to this functionality as Haskells hidden powers.

In this work, we expose the following three hidden powers to Haskell programmers in a safe way. These features were previously unavailable or only available with less safety guarantees. The first feature is partial type signatures, which exposes the hidden power of unification variables. Secondly, coherent explicit dictionary application exposes the hidden power of the type class dictionaries. Finally, the third feature is expressive and strongly type-safe code generation, which safely exposes more of the hidden power of compile-time code generation. All three features were previously available only in unsafe or less expressive forms. We propose their design, provide formal correctness arguments and have implemented them (in at least a prototype form) for the GHC compiler.