JFP **25**, e1, 16 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S0956796815000040

PhD Abstracts

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

Many students complete PhDs in functional programming each year, but there is currently no common location in which to promote and advertise the resulting work. The Journal of Functional Programming would like to change that!

As a service to the community, JFP recently launched a new feature, in the form of a regular publication of abstracts from PhD dissertations that were completed during the previous year. The abstracts are made freely available on the JFP website, i.e. not behind any paywall, and do not require any transfer for copyright, merely a license from the author. A dissertation is eligible 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 second 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 editor for further details.

Graham Hutton
PhD Abstract Editor

Erasable coercions: A unified approach to type systems

JULIEN CRETIN
Université Paris-Diderot - Paris VII, Paris, France

Date: January 2014; Advisor: Didier Rémy
URL: `http://tel.archives-ouvertes.fr/tel-00940511`

Functional programming languages, like OCaml or Haskell, rely on the lambda calculus for their core language. Although they have different reduction strategies and type system features, their proof of soundness and normalization (in the absence of recursion) should be factorizable. This thesis does such a factorization for theoretical type systems featuring recursive types, subtyping, bounded polymorphism and constraint polymorphism. Interestingly, soundness and normalization for strong reduction imply soundness and normalization for all usual strategies. Our observation is that a generalization of existing coercions permits to describe all type system features stated above in an erasable and composable way. We illustrate this by proposing two concrete type systems: first, an explicit type system with a restricted form of coercion abstraction to express subtyping and bounded polymorphism; and an implicit type system with unrestricted coercion abstraction that generalizes the explicit type system with recursive types and constraint polymorphism – but without the subject reduction property. A side technical result is an adaptation of the step-indexed proof technique for type-soundness to calculi equipped with a strong notion of reduction.

Functional techniques for representing and specifying software

DOMINIQUE DEVRIESE
Katholieke Universiteit Leuven, Leuven, Belgium

Date: January 2014; Advisor: Frank Piessens
URL: `https://lirias.kuleuven.be/bitstream/123456789/425797/1/thesis.pdf`

All softwares are represented as source code in a programming language. The programming language defines the meaning or semantics of the code, for example, its operational behaviour. Computational source code is often accompanied by additional specifications that define how the source code should be interpreted or provide additional information about the software's semantics. They make it possible for programmers to express and verify that their software has the intended semantics and to express inter-component semantic assumptions. Good representations and specifications of software components are crucial for efficiently producing software that is reliable, efficient and secure, and for preserving these qualities during the software's evolution. Many types of software components and their desired semantic properties can be challenging to represent and specify. In this work, I contribute novel functional techniques for the representation and specification of four types of software components which are as follows:

Ad hoc polymorphic functions: Functions whose behaviour depends on the types of their arguments or result. I present instance arguments: a type system extension for representing ad hoc polymorphic functions in the dependently-typed programming language Agda. Compared to existing proposals, instance arguments do not introduce an additional structuring concept and ad hoc polymorphic functions using them are fully first-class. Furthermore, they avoid introducing a separate, powerful form of type-level computation and existing Agda libraries using records do not need modifications to be used with them. My implementation has been part of Agda since version 2.3.0 and I demonstrate a variety of applications of instance arguments.

Context-free grammars: A standard way to define the syntax of formal languages. I present a technique for representing context-free grammars in an embedded domain-specific language. It avoids the restrictions of existing parser combinator libraries using a novel explicit representation of recursion based on advanced type system techniques in the Haskell programming language. As a byproduct, grammars are decoupled from sets of semantic actions. On the flip side, the approach requires the grammar's author to provide a type- and value-level encoding of the grammar's domain and I can provide only a limited form of constructs like many. I demonstrate the approach with five grammar algorithms, including a pretty-printer, a reachability analysis, a translation of quantified recursive constructs to standard ones, and an implementation of the left-corner grammar transform. This work forms the basis of my grammar-combinators parsing library.

Meta-programs: Programs that generate or manipulate other programs. I present a novel set of meta-programming primitives for use in a dependently-typed functional language. The meta-programs' types provide strong and precise guarantees about the meta-programs' termination, correctness and completeness. The system supports type-safe construction and analysis of terms, types and typing contexts. Unlike alternative approaches, meta-programs are written in the same style as normal programs and use the language's standard functional computational model. I formalize the new meta-programming primitives, implement them as an extension of Agda, and provide evidence of usefulness by means of two compelling applications in the fields of datatype-generic programming and proof tactics.

Effect polymorphic software: Programs that support arbitrary implementations of effectful Application Program Interface (APIs) and only produce effects through those implementations. Static effectful APIs and global mutable state in object-oriented programming languages make it hard to modularly control effects. Object-capability (OC) languages solve this by enforcing that effects can only be triggered by components that hold a reference to the object representing the capability to do so. I study this encapsulation of effects through a formal translation to a typed functional calculus with higher-ranked polymorphism (I use a subset of Haskell for presentation). Based on an informal view of effect-polymorphism as the fundamental feature of OC languages, I translate an OC calculus to effect-polymorphic Haskell code, i.e. computations that are universally quantified over the monad in which they produce effects. The types of my translations assert the OC property and I can show and exploit this using Reynolds' parametricity theorem. An important new insight is that current OC languages and formalisations leave one effect implicitly available to all code, without a capability: the allocation of new mutable state; adding a capability for it has important theoretical and practical advantages. My work establishes a new link between OC languages and the well-studied fields of functional programming and denotational semantics.

Effective aspects: A typed monadic model to control and reason about aspect interference

ISMAEL FIGUEROA
Universidad de Chile, Chile and École des Mines de Nantes, France

Date: April 2014; Advisor: Éric Tanter, Nicolas Tabareau and Mario Südholt
URL: `http://www.dart-europe.eu/full.php?id=1010984`

Aspect-oriented programming (AOP) aims to enhance modularity and reusability in software systems by offering an abstraction mechanism to deal with crosscutting concerns. However, in most general-purpose aspect languages aspects have almost unrestricted power, eventually conflicting with these goals. In this work, we present EffectiveAspects: a novel approach to embed the pointcut/advice model of aspect-oriented programming in a statically-typed functional programming language like Haskell. Our work comprises two main contributions.

First, we define a monadic embedding of the full pointcut/advice model of aspect-oriented programming. Type soundness is guaranteed by exploiting the underlying type system, in particular phantom types and a new anti-unification type class. In this model aspects are first-class, can be deployed dynamically, and the pointcut language is extensible, therefore combining the flexibility of dynamically-typed aspect languages with the guarantees of a static type system. Monads enable us to directly reason about computational effects both in aspects and base programs using traditional monadic techniques. Using this we extend Aldrich's notion of open modules with effects, and also with protected pointcut interfaces to external advising. These restrictions are enforced statically using the type system. Also, we adapt the techniques of EffectiveAdvice, a mixin-based model of advice developed by Schrijvers, Oliveira and Cook, to reason about and enforce control flow properties; and to control effect interference by restricting access to the different layers of the monad stack, using a parametricity-based approach. Moreover, we illustrate that the parametricity-based approach falls short in the presence of multiple aspects and propose a different approach using *monad views*, a novel technique for handling the monad stack, developed by Schrijvers and Oliveira. Then, we exploit the properties of our model to enable the modular construction of new semantics for aspect scoping and weaving.

Our second contribution builds upon MRI, which stands for *modular reasoning for incremental programming*, a powerful model that extends EffectiveAdvice in order to reason about mixin-based composition of effectful components and their interference, based on equational reasoning, parametricity and algebraic laws about monadic effects. Our contribution is to show how to reason about interference in the presence of unrestricted quantification through pointcuts. We show that global reasoning can be compositional, which is a key for the scalability of the approach in the face of large and evolving systems. We prove a general equivalence theorem that is based on a few conditions that can be established, reused and adapted

separately as the system evolves. The theorem is defined for an abstract monadic, aspect-oriented programming model; we illustrate its use with a simple version of the model just described. This work brings type-based reasoning about effects for the first time in the pointcut/advice model, in a framework that is both expressive and extensible. The framework is well-suited for development of robust aspect-oriented systems as well as being a research tool for experimenting and reasoning about new aspect semantics.

Analysis and synthesis of inductive families

HSIANG-SHANG KO
University of Oxford, Oxford, UK

Date: September 2014; Advisor: Jeremy Gibbons
URL: `http://ora.ox.ac.uk/objects/uuid:2bc39bde-ce59-4a49-b499-3afdf174bbab`

Based on a natural unification of logic and computation, Martin-Löf's *intuitionistic type theory* can be regarded simultaneously as a computationally meaningful higher-order logic system and an expressively typed functional programming language, in which proofs and programs are treated as the same entities. Two modes of programming can then be distinguished: in *externalism*, we construct a program separately from its correctness proof with respect to a given specification, whereas in *internalism*, we encode the specification in a sophisticated type such that any program inhabiting the type also encodes a correctness proof, and we can use type information as a guidance on program construction. Internalism is particularly effective in the presence of *inductive families*, whose design can have a strong influence on program structure. Techniques and mechanisms for facilitating internalist programming are still lacking, however.

This dissertation proposes that internalist programming can be facilitated by exploiting an interconnection between internalism and externalism, expressed as isomorphisms between inductive families into which data structure invariants are encoded and their simpler variants paired with predicates expressing those invariants. The interconnection has two directions: one *analysing* inductive families into simpler variants and predicates, and the other *synthesising* inductive families from simpler variants and specific predicates. They respectively give rise to two applications, one achieving a modular structure of internalist libraries, and the other bridging internalist programming with relational specifications and program derivation. The datatype-generic mechanisms supporting the applications are based on McBride's *ornaments*. Theoretically, the key ornamental constructs – *parallel composition of ornaments* and *relational algebraic ornamentation* – are further characterised in terms of lightweight category theory. Most of the results are completely formalised in the Agda programming language.

Extending implicit computational complexity and abstract machines to languages with control

GIULIO PELLITTA
University of Bologna, Bologna, Italy

Date: May 2014; Advisor: Simone Martini
URL: http://amsdottorato.unibo.it/6558/

Introduction: The Curry–Howard isomorphism is the idea that proofs in natural deduction can be put in correspondence with lambda terms in such a way that this correspondence is preserved by normalization. The concept can be extended from intuitionistic logic to other systems, such as linear logic. One of the nice consequences of this isomorphism is that we can reason about functional programs with formal tools which are typical of proof systems: such analysis can also include quantitative qualities of programs, such as the number of steps it takes to terminate. Another is the possibility to describe the execution of these programs in terms of abstract machines. In 1990 Griffin proved that the correspondence can be extended to Classical Logic and *control operators*. That is, Classical Logic adds the possibility to manipulate *continuations*. In this thesis we see how the things we described above work in this larger context.

Outline: In Part I, we analyze some variants of lambda mu-calculus (an extension of lambda calculus with two control operators for manipulating continuations), with special attention to the calculus of de Groote; and the main ideas of implicit computational complexity (a series of tools and techniques to characterize complexity classes), especially restrictions of linear logic with limited complexity and in particular bounded linear logic (BLL). In Part II, we investigate how to adapt techniques of implicit computational complexity to characterize a fragment of lambda mu-calculus which is expressive enough to represent all functions that can be computed in time polynomial w.r.t. the input size. In Part III, we study some abstract machines for evaluating lambda- and lambda mu-terms; we ultimately review an abstract machine for lambda terms which has not received a lot of attention in the last few years and adapt it to lambda mu-calculus.

Part I: We introduce the main subjects of the thesis, namely control operators and implicit computational complexity. Continuations can enrich functional programming languages with imperative features. We review the operators call/cc (call-with-current-continuation) and Felleisen's C, which have been extensively studied in the literature. Finally, we introduce Parigot's lambda mu-calculus, which extends lambda calculus with two operators called mu and bracket, and then the extended calculus by De Groote, in which the operators mu and bracket do not have to be paired. Implicit computational complexity aims at studying machine-free characterization of complexity classes with a mathematically-oriented approach. Linear logic, in which duplication/erasure are explicit, has proved of fundamental importance in this field. Among various type systems for lambda calculus inspired by variants of linear logic with restricted rules, we focus on Girard's BLL, a type systems for terms that can be run in polynomial time w.r.t. input size.

Part II: Polarized bounded linear logic is defined starting from Laurent's work on polarized linear logic, a variant of linear logic, which can be used as a type system for lambda mu, where contraction can be applied to any *negative formula*. Formulas of polarized bounded linear logic can be either positive or negative just likein polarized linear logic, but have exponential modalities which are bounded as in BLL. Polarized bounded linear logic can be used as type system for De Groote's lambda mu which characterizes terms which can be evaluated in *polynomial time* (analogously to BLL). The system is expressive enough to type the aforementioned call/cc and Felleisen's C. One significant improvement w.r.t. BLL is that the polynomial-soundness result holds w.r.t. head reduction (that is, we are allowed to reduce under lambda and mu binders).

Part III: The Krivine abstract machine is a mechanism for call-by-name evaluation of lambda terms (which has been extended by De Groote for lambda mu-calculus). The Krivine abstract machine computes the (weak) head normal form of a lambda term, performing a series of transitions depending on the shape of the term (more precisely, it performs head *linear* reduction). It is based on the notion of *closure*, a pair consisting of a term and a sequence of closures. The main drawback of the Krivine abstract machine is that it may build closures that are never used during execution. One variant of the Krivine abstract machine designed to avoid this, namely the Pointer Abstract Machine (PAM), has been studied by Herbelin, Danos and Regnier. Instead of working with closures, it uses *pointers* to keep track of where a closure should have been build so that the corresponding term can be retrieved on the fly. The PAM is explained somewhat more clearly than before (e.g., using explicit substitutions rather than game semantics, in a more operational approach) and it is suggested how it can be adapted to lambda mu-calculus.

Operational aspects of full reduction in lambda calculi

ÁLVARO GARCÍA PÉREZ
Universidad Politécnica de Madrid, Madrid, Spain

Date: October 2014; Advisor: Juan José Moreno Navarro and Pablo Nogueira
URL: `http://babel.ls.fi.upm.es/ agarcia/phd-thesis/thesis.pdf`

This thesis studies full reduction in lambda calculi. In a nutshell, full reduction consists in evaluating the body of the functions in a functional programming language with binders. The classical (i.e. pure untyped) lambda calculus is set as the formal system that models the functional paradigm. Full reduction is a prominent technique when programs are treated as data objects, for instance when performing optimisations by partial evaluation, or when some attribute of the program is represented by a program itself, like the type in modern proof assistants.

A notable feature of many full-reducing operational semantics is its hybrid nature, which is introduced and which constitutes the guiding theme of the thesis. In the lambda calculus, the hybrid nature amounts to a "phase distinction" in the treatment of abstractions when considered either from outside or from inside themselves. This distinction entails a layered structure in which a hybrid semantics depends on one or more subsidiary semantics.

From a programming languages standpoint, the thesis shows how to derive implementations of full-reducing operational semantics from their specifications, by using program transformations techniques. The program transformation techniques are syntactical transformations which preserve the semantic equivalence of programs. The existing program transformation techniques are adjusted to work with implementations of hybrid semantics. The thesis also shows how full reduction impacts the implementations that use the environment technique. The environment technique is a key ingredient of real-world implementations of abstract machines which helps to circumvent the issue with binders.

From a formal systems standpoint, the thesis discloses a novel consistent theory for the call-by-value variant of the lambda calculus which accounts for full reduction. This novel theory entails a notion of observational equivalence which distinguishes more points than other existing theories for the call-by-value lambda calculus. This contribution helps to establish a "standard theory" in that calculus which constitutes the analogous of the "standard theory" advocated by Barendregt in the classical lambda calculus. Some proof-theoretical results are presented, and insights on the model-theoretical study are given.

Certificates for incremental type checking

MATTHIAS PUECH
Università di Bologna, Italy and Université Paris Diderot - Paris VII, France

Date: April 2013; Advisor: Andrea Asperti and Hugo Herbelin
URL: `http://amsdottorato.unibo.it/5870/`

The central topic of this thesis is the study of algorithms for type checking, both from the programming language and from the proof-theoretic point of view. A type checking algorithm takes a program or a proof, represented as a syntactical object, and checks its validity with respect to a specification or a statement; it is a central piece of compilers and proof assistants. First, we present a tool which supports the development of functional programs manipulating proof certificates (certifying programs). It uses LF as a representation metalanguage for higher-order proofs and OCaml as a programming language, and facilitates the automated and efficient verification of these certificates at run time. Technically, we introduce in particular the notion of function inverse allowing to abstract from a local environment when manipulating open terms. Then, we remark that the idea of a certifying type checker, generating a typing derivation, can be extended to realize an incremental type checker, working by reuse of typing subderivation. Such a type checker would make possible the structured and type-directed edition of proofs and programs. Finally, we showcase an original correspondence between natural deduction and the sequent calculus, through the transformation of the corresponding type checking functional programs: we show, using off-the-shelf program transformations, that the latter is the accumulator-passing version of the former.

Intersection types and higher-order model checking

STEVEN J. RAMSAY
University of Oxford, Oxford, UK

Date: August 2014; Advisor: C.-H. Luke Ong
URL: `http://ora.ox.ac.uk/objects/uuid:46b7bc70-3dfe-476e-92e7-245b7629ae4e`

Higher-order recursion schemes are systems of equations that are used to define finite and infinite labelled trees. Since, as Ong has shown, the trees defined have a decidable monadic second order theory, recursion schemes have drawn the attention of research in program verification, where they sit naturally as a higher-order, functional analogue of Boolean programs. Driven by applications, fragments have been studied, algorithms developed and extensions proposed; the emerging theme is called higher-order model checking. Kobayashi has pioneered an approach to higher-order model checking using intersection types, from which many recent advances have followed. The key is a characterisation of model checking as a problem of intersection type assignment. This dissertation contributes to both the theory and practice of the intersection type approach.

A new, fixed-parameter polynomial-time decision procedure is described for the alternating trivial automaton fragment of higher-order model checking. The algorithm uses a novel, type-directed form of abstraction refinement, in which behaviours of the scheme are distinguished according to the intersection types that they inhabit. Furthermore, by using types to reason about acceptance and rejection simultaneously, the algorithm is able to converge on a solution from two sides. An implementation, preface, and an extensive body of evidence demonstrate empirically that the algorithm scales well to schemes of several thousand rules. A comparison with other tools on benchmarks derived from current practice and the related literature puts it well beyond the state-of-the-art.

A generalisation of the intersection type approach is presented in which higher-order model checking is seen as an instance of exact abstract interpretation. Intersection type assignment is used to characterize a general class of safety checking problems, defined independently of any particular representation (such as automata) for a class of recursion schemes built over arbitrary constants. Decidability of any problem in the class is an immediate corollary. Moreover, the work looks beyond whole-program verification, the traditional territory of model checking, by giving a natural treatment of higher-type properties, which are sets of functions.

Free theorems in languages with real-world programming features

DANIEL SEIDEL
Rheinische Friedrich-Wilhelms-Universität Bonn, Bonn, Germany

Date: October 2013; Advisor: Janis Voigtländer
URL: `http://nbn-resolving.de/urn:nbn:de:hbz:5n-33798`

Free theorems, type-based assertions about functions, have become a prominent reasoning tool in functional programming languages. But their correct application requires a lot of care. Restrictions arise due to features present in implemented such languages, but not in the language free theorems were originally investigated in.

This thesis advances the formal theory behind free theorems w.r.t. the application of such theorems in non-strict functional languages such as Haskell. In particular, the impact of general recursion and forced strict evaluation is investigated. As formal ground, we employ different lambda calculi equipped with a denotational semantics.

For a language with general recursion, we develop and implement a counterexample generator that tells if and why restrictions on a certain free theorem arise due to general recursion. If a restriction is necessary, the generator provides a counterexample to the unrestricted free theorem. If not, the generator terminates without returning a counterexample. Thus, we may on the one hand enhance the understanding of restrictions and on the other hand, point to cases where restrictions are superfluous.

For a language with a strictness primitive, we develop a refined type system that allows to localize the impact of forced strict evaluation. Refined typing results in stronger free theorems and therefore increases the value of the theorems. Moreover, we provide a generator for such stronger theorems.

Lastly, we broaden the view on the kind of assertions free theorems provide. For a very simple, strict evaluated, calculus, we enrich free theorems by (runtime) efficiency assertions. We apply the theory to several toy examples. Finally, we investigate the performance gain of the foldr/build program transformation. The latter investigation exemplifies the main application of our theory: free theorems may not only ensure semantic correctness of program transformations, they may also ensure that a program transformation speeds up a program.

Incremental parallelization of existing sequential runtime systems

JAMES EDWARD SWAINE
Northwestern University, Evanston, IL, USA

Date: June 2014; Advisor: Robert Bruce Findler
URL: `http://plt.eecs.northwestern.edu/swaine-phd.pdf`

Many language implementations, particularly for high-level and scripting languages, are based on carefully honed runtime systems that have an internally sequential execution model. Adding support for parallelism in the usual form – as threads that run arbitrary code in parallel – would require a major revision or even a rewrite to add safe and efficient locking and communication. This dissertation describes an alternative approach to *incremental parallelization* of runtime systems. This approach can be applied inexpensively to many sequential runtime systems, and we demonstrate its effectiveness in the Racket runtime system and Parrot virtual machine. The evaluation assesses performance benefits, developer effort needed to implement such a system in these two runtime systems, and the ease with which users can leverage the resulting parallel programming constructs without sacrificing expressiveness. We find that incremental parallelization can provide useful, scalable parallelism on commodity multicore processors at a fraction of the effort required to implement conventional parallel threads.

The choice calculus: A formal language of variation

ERIC WALKINGSHAW
Oregon State University, Corvallis, OR, USA

Date: June 2013; Advisor: Martin Erwig
URL: `http://hdl.handle.net/1957/40652`

In this thesis I present the choice calculus, a formal language for representing variation in software and other structured artefacts. The choice calculus is intended to support variation research in a way similar to the lambda calculus in programming language research. Specifically, it provides a simple formal basis for presenting, proving and communicating theoretical results. It can serve as a common language of discourse for researchers working on different views of similar problems and provide a shared back-end in tools.

This thesis collects a large amount of work on the choice calculus. It defines the syntax and denotational semantics of the language along with modular language extensions that add features important to variation research. It presents several theoretical results related to the choice calculus, such as an equivalence relation that supports semantics-preserving transformations of choice calculus expressions, and a type system for ensuring that an expression is well formed. Many of these results have been reused successfully in our work on extending Hindley–Milner type inference to variational programs.

This thesis also presents a domain-specific language embedded in Haskell, based on the choice calculus, for exploring the concept of variational programming.

The interpretation and inter-derivation of small-step and big-step specifications

IAN ZERNY
Aarhus University, Aarhus, Denmark

Date: June 2013; Advisor: Olivier Danvy and Jan Midtgaard
URL: `http://www.zerny.dk/zerny-phd.html`

We study the interpretation and inter-derivation of *big-step* and *small-step* specifications. In particular, we consider formal specifications of programming languages, e.g., denotational semantics and operational semantics, and investigate how these specifications relate to each other. We carry out this investigation by interpreting specifications as programs in a pure functional metalanguage and by constructively deriving one program from the other using program transformations. To this end, we use two derivational correspondences: the *functional correspondence* between compositional higher-order specifications and first-order transition systems, and the *syntactic correspondence* between rewriting specifications and first-order transition systems.

The main contribution of this dissertation is threefold: first, we extend these correspondences to systematically derive small-step reduction semantics and abstract machines from big-step reduction strategies. Second, we show how these correspondences can be used to relate specifications for lazy evaluation, e.g., graph reduction and call-by-need evaluation. Third, we describe an alternative interpretation of specifications as logic programs in a logical framework, and we give a logical counterpart to the functional correspondence.