JFP **25**, e7, 9 pages, 2015.
doi:10.1017/S0956796815000167

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 8 abstracts for 2014/15 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

Terrier: An Embedded Operating System Using Advanced Types for Safety

MATTHEW DANISH
Boston University, USA

Date: May 2015; Advisor: Hongwei Xi
URL: http://cs-people.bu.edu/md/dissertation/md.pdf

Operating systems software is fundamental to modern computer systems: all other applications are dependent upon the correct and timely provision of basic system services. At the same time, advances in programming languages and type theory have lead to the creation of functional programming languages with type systems that are designed to combine theorem proving with practical systems programming. The Terrier operating system project focuses on low-level systems programming in the context of a multi-core, real-time, embedded system, while taking advantage of a dependently typed programming language named ATS to improve reliability. Terrier is a new point in the design space for an operating system, one that leans heavily on an associated programming language, ATS, to provide safety that has traditionally been in the scope of hardware protection and kernel privilege. Terrier tries to have far fewer abstractions between program and hardware. The purpose of Terrier is to put programs as much in contact with the real hardware, real memory, and real timing constraints as possible, while still retaining the ability to multiplex programs and provide for a reasonable level of safety through static analysis.

On Computational Small Steps and Big Steps: Refocusing for Outermost Reduction

JACOB JOHANNSEN
Aarhus University, Denmark

Date: March 2015; Advisor: Olivier Danvy
URL: http://pure.au.dk/portal/files/86391423/Jacob_Johannsen_dissertation.pdf

We study the relationship between small-step semantics, big-step semantics and abstract machines, for programming languages that employ an outermost reduction strategy, i.e., languages where reductions near the root of the abstract syntax tree are performed before reductions near the leaves. In particular, we investigate how Biernacka and Danvy's syntactic correspondence and Reynolds's functional correspondence can be applied to inter-derive semantic specifications for such languages.

The main contribution of this dissertation is three-fold: First, we identify that backward overlapping reduction rules in the small-step semantics cause the refocusing step of the syntactic correspondence to be inapplicable. Second, we propose two solutions to overcome this in-applicability: *backtracking* and *rule generalization*. Third, we show how these solutions affect the other transformations of the two correspondences.

Other contributions include the application of the syntactic and functional correspondences to Boolean normalization. In particular, we show how to systematically derive a spectrum of normalization functions for negational and conjunctive normalization.

Automating Abstract Interpretation of Abstract Machines

JAMES IAN JOHNSON
Northeastern University, USA

Date: April 2015;
Advisor: David Van Horn
URL: http://arxiv.org/abs/1504.08033

Static program analysis is a valuable tool for any programming language that people write programs in. The prevalence of scripting languages in the world suggests programming language interpreters are relatively easy to write. Users of these languages lament their inability to analyze their code, therefore programming language analyzers (abstract interpreters) are not easy to write. This thesis more deeply investigates a systematic method of creating abstract interpreters from traditional interpreters, called Abstracting Abstract Machines.

Abstract interpreters are difficult to develop due to technical, theoretical, and pragmatic problems. Technical problems include engineering data structures and algorithms. I show that modest and simple changes to the mathematical presentation of abstract machines result in 1000 times better running time - just seconds for moderately sized programs.

In the theoretical realm, abstraction can make correctness difficult to ascertain. Analysis techniques need a reason to trust them. Previous analysis techniques, if they have a correctness proof, will have to bridge multiple formulations of a language's semantics to prove correct. I provide proof techniques for proving the correctness of regular, pushdown, and stack-inspecting pushdown models of abstract computation by leaving computational power to an external factor: allocation. Each model is equivalent to the concrete (Turing-complete) semantics when the allocator creates fresh addresses. Even if we don't trust the proof, we can run models concretely against test suites to better trust them. If the allocator reuses addresses from a finite pool, then the structure of the semantics collapses to one of these three sound automata models, without any foray into automata theory.

In the pragmatic realm, I show that the systematic process of abstracting abstract machines is automatable. I develop a meta-language for expressing abstract machines similar to other semantics engineering languages. The language's special feature is that it provides an interface to abstract allocation. The semantics guarantees that if allocation is finite, then the semantics is a sound and computable approximation of the concrete semantics. I demonstrate the language's expressiveness by formalizing the semantics of a Scheme-like language with temporal higher-order contracts, and automatically deriving a computable abstract semantics for it.

Programming Contextual Computations

DOMINIC ORCHARD
University of Cambridge, UK

Date: May 2014; Advisor: Alan Mycroft
URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-854.html

Modern computer programs are executed in a variety of different *contexts*: on servers, handheld devices, graphics cards, and across distributed environments, to name a few. Understanding a program's *contextual* requirements is therefore vital for its correct execution. This dissertation studies contextual computations, ranging from application-level notions of context to lower-level notions of context prevalent in common programming tasks. It makes contributions in three areas: mathematically structuring contextual computations, analysing contextual program properties, and designing languages to facilitate contextual programming.

Firstly, existing work which mathematically structures contextual computations using *comonads* (in programming and semantics) is analysed and extended. Comonads are shown to exhibit a *shape preservation* property which restricts their applicability to a subset of contextual computations. Subsequently, novel generalisations of comonads are developed, including the notion of an *indexed comonad*, relaxing shape-preservation restrictions.

Secondly, a general class of static analyses called *coeffect* systems is introduced to describe the propagation of contextual requirements throughout a program. Indexed comonads, with some additional structure, are shown to provide a semantics for languages whose contextual properties are captured by a coeffect analysis.

Finally, language constructs are presented to ease the programming of contextual computations. The benefits of these language features, the mathematical structuring, and coeffect systems are demonstrated by a language for container programming which guarantees optimisations and safety invariants.

Random Structured Test Data Generation for Black-Box Testing

MICHAŁ H. PAŁKA
Chalmers University of Technology, Sweden

Date: December 2014; Advisor: Koen Claessen and John Hughes
URL: http://tinyurl.com/pw9vkru

We show how automated random testing can be used to effectively find bugs in complex software, such as an optimising compiler. To test the GHC Haskell compiler we created a generator of simple random programs, used GHC to compile them with different optimisation levels, and then compared the results of running them. Using this simple approach we found a number of optimisation bugs in GHC. This approach for finding bugs proved to be very effective, but we found that implementing a generator of random programs by hand required a large amount of effort. Therefore, we developed an automatic method for deriving random generators of complex test data based on computable boolean predicates that specify the well-formed values of the data type. Defining such a predicate is usually much quicker than implementing a dedicated generator, even if its performance might be comparably lower. In addition, we discovered that the pseudorandom number generator used by us for random testing is unreliable, and that no reliable contruction exists that supports our particular requirements. Consequently, we designed and implemented a high-quality pseudorandom number generator, which is based on a known and reliable cryptographic construction, and whose correctness is supported by a formal argument. Finally, we present how random testing can be used to rank a group of programs according to their relative correctness with respect to their observed behaviour. The ranking method removes the influence of the distribution of the random data generator used for testing, which results in a reliable ranking.

Profiling Optimised Haskell – Causal Analysis and Implementation

PETER MORITZ WORTMANN
University of Leeds, UK

Date: October 2014; Advisor: David Duke
URL: http://etheses.whiterose.ac.uk/8321/

At the present time, performance optimisation of real-life Haskell programs is a bit of a "black art". Programmers that can do so reliably are highly esteemed, doubly so if they manage to do it without sacrificing the character of the language by falling back to an "imperative style". The reason is that while programming at a high-level does not need to result in slow performance, it must rely on a delicate mix of optimisations and transformations to work out just right. Predicting how all these cogs will turn is hard enough - but where something goes wrong, the various transformations will have mangled the program to the point where even finding the crucial locations in the code can become a game of cat-and-mouse.

In this work we will lift the veil on the performance of heavily transformed Haskell programs: Using a formal causality analysis we will track source code links from square one, and maintain the connection all the way to the final costs generated by the program. This will allow us to implement a profiling solution that can measure performance at high accuracy while explaining in detail how we got to the point in question. Furthermore, we will directly support the performance analysis process by developing an interactive profiling user interface that allows rapid theory forming and evaluation, as well as deep analysis where required.

Combinatorial Species and Labelled Structures

BRENT A. YORGEY
University of Pennsylvania, USA

Date: December 2014;
Advisor: Stephanie Weirich
URL: `http://tinyurl.com/o7d6xue`

The theory of *combinatorial species* was developed in the 1980s as part of the mathematical subfield of enumerative combinatorics, unifying and putting on a firmer theoretical basis a collection of techniques centered around *generating functions*. The theory of *algebraic data types* was developed, around the same time, in functional programming languages such as Hope and Miranda, and is still used today in languages such as Haskell, the ML family, and Scala. Despite their disparate origins, the two theories have striking similarities. In particular, both constitute algebraic frameworks in which to construct structures of interest. Though the similarity has not gone unnoticed, a link between combinatorial species and algebraic data types has never been systematically explored. This dissertation lays the theoretical groundwork for a preciseand, hopefully, usefulbridge bewteen the two theories. One of the key contributions is to port the theory of species from a classical, untyped set theory to a constructive type theory. This porting process is nontrivial, and involves fundamental issues related to equality and finiteness; the recently developed *homotopy type theory* is put to good use formalizing these issues in a satisfactory way. In conjunction with this port, species as general functor categories are considered, systematically analyzing the categorical properties necessary to define each standard species operation. Another key contribution is to clarify the role of species as *labelled shapes*, not containing any data, and to use the theory of *analytic functors* to model labelled data structures, which have both labelled shapes and data associated to the labels. Finally, some novel species variants are considered, which may prove to be of use in explicitly modelling the memory layout used to store labelled data structures.

Interactive Typed Tactic Programming in the Coq Proof Assistant

BETA ZILIANI
Saarland University, Germany

Date: March 2015; Advisor: Derek Dreyer
URL: http://scidok.sulb.uni-saarland.de/volltexte/2015/6041/pdf/thesis.pdf

In order to allow for the verification of realistic problems, Coq provides a language for *tactic* programming, therefore enabling general-purpose scripting of automation routines. However, this language is untyped, and as a result, tactics are known to be difficult to compose, debug, and maintain. In this thesis, I develop two different approaches to typed tactic programming in the context of Coq: *Lemma Overloading* and *Mtac*. The first one utilizes the existing mechanism of overloading, already incorporated into Coq, to build typed tactics in a style that resembles that of dependently typed *logic* programming. The second one, Mtac, is a lightweight yet powerful extension to Coq that supports dependently typed *functional* tactic programming, with additional imperative features.

I motivate the different characteristics of Lemma Overloading and Mtac through a wide range of examples, mainly coming from program verification. I also show how to combine these approaches in order to obtain the best of both worlds, resulting in *extensible*, typed tactics that can be programmed interactively.

Both approaches rely heavily on the unification algorithm of Coq, which currently suffers from two main drawbacks: it incorporates heuristics not appropriate for tactic programming, and it is undocumented. In this dissertation, in addition to the aforementioned approaches to tactic programming, I build and describe a new unification algorithm better suited for tactic programming in Coq.