JFP **31**, e14, 15 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S0956796821000149

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 fourteen 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

Do-it-Yourself Module Systems: Extending Dependently-Typed Languages to Implement Module System Features In The Core Language

MUSA AL-HASSY
McMaster University, Canada

Date: April 2021; Advisor: Wolfram Kahl and Jacques Carette
URL: https://tinyurl.com/2cuntnb5

We show that contexts serve as a *practical* single-source of truth from which one may extract other useful packaging constructs such as records, typeclasses, algebraic data-types, and much more. Along the way we solve The Unbundling Problem.

Imagine you have a collection of physical folders, portfolios, of candidates to hire at your workplace. You may be interested in *only looking at* their experience in the field before looking at the rest of their portfolio, so you request a secretary to produce new folders that have the numeric year on the front and all other information within the folder. You can now quickly discern which candidates are interesting to you. However, a colleague may be interested in *only looking at* their publication count. Once again, the secretary will construct an entire new set of portfolios, *restructured to meet the needs of a new user*. Yet another hiring manager may be interested in *only looking at* their current residing location *and* their school of study; and so the secretary is summoned yet again. The ability to *expose* certain details of a candidate's portfolio to the top cover of the folder is useful to *narrow focus* to what is important to know first, before any further consideration. However, what is 'important' is different to different people.

We show how to solve this exposing, "unbundling", problem in dependently-typed languages with a *pragmatic, usable,* implementation in Agda —as well as a more powerful implementation in Emacs Lisp, serving as an "editor extension".

Put simply, the thesis is about making *tedious and inexpressible patterns* of programming in dependently typed languages become *mechanical and expressible*. In particular, we show how Σ, Π, and $\mathscr{W}$ types can be obtained from contexts, using monadic do-notation. As such, one avoids the repetitious and error-prone activity of forming similar, related, grouping mechanisms —and their coercions— manually, on a case-by-case basis. The monadic definition of contexts allows users to "extend" the system to allow for unexpected interpretations of contexts —using the same concrete syntax.

On the Foundations of Practical Language-Based Security

MAXIMILIAN ALGEHED
Chalmers University of Technology, Sweden

Date: June 2021; Advisor: Mary Sheeran and Koen Claessen
URL: https://tinyurl.com/543bfpc5

Language-based information flow control (IFC) promises to provide programming languages and tools that make it easy for developers to write secure code. Traditionally, one builds a variant of a programming language or system that lets developers write code with strong security guarantees. However, two developments challenge this paradigm. Firstly, backwards-compatible security enforcement without false alarms promises to retrofit security in code that was not written with this in mind. Secondly, library-based security promises to do away with specialized IFC languages by factoring them into libraries.

This thesis makes contributions to both these developments that come in two parts; the first concerns enforcing secure information flow without false alarms while the second concerns the correctness of IFC libraries.

The first part makes the following contributions:

1. It unifies the existing literature on security enforcement without false alarms by introducing Faceted Secure Multi-Execution.
2. It explores the unique optimisation challenges that appear in this setting.
3. It proves an exponential lower bound on black-box false-alarm-free enforcement and new possibility results for false-alarm-free enforcement.
4. It classifies the special cases of enforcement that is not subject to the aforementioned exponential lower bound.

In short, the first part of the thesis unifies the existing literature on false-alarm-free IFC enforcement and presents a number of results on the performance of enforcement mechanisms of this kind.

The second part meanwhile makes the following contributions:

1. It reduces the complexity of security libraries by showing how to implement secure effects on top of an already secure core.
2. It shows how to simplify DCC, the core language in the literature, without losing expressiveness.
3. It proves that noninterference can be derived in a straightforward way from parametricity for both static and dynamic security libraries. This reduces the gap between the security libraries that are used today and the proofs used to prove that the libraries ensure noninterference.

In short, the second part of the thesis provides a new direction for thinking about the correctness of security libraries by both reducing the amount of trusted code and by introducing improved means of proving that a security library guarantees noninterference.

Towards Efficient Gradual Typing via Monotonic References and Coercions

DEYAAELDEEN ALMAHALLAWI
Indiana University Bloomington, USA

Date: June 2020; Advisor: Jeremy G. Siek
URL: https://tinyurl.com/udzbemk3

Integrating static and dynamic typing into a single programming language enables programmers to choose which discipline to use in each code region. Different approaches for this integration have been studied and put into use at large scale, e.g. TypeScript for JavaScript and adding the dynamic type to C#. Gradual typing is one approach to this integration that preserves type soundness by performing type-checking at run-time using casts. For higher order values such as functions and mutable references, a cast typically wraps the value in a proxy that performs type-checking when the value is used. This approach suffers from two problems: (1) chains of proxies can grow and consume unbounded space, and (2) statically typed code regions need to check whether values are proxied. Monotonic references solve both problems for mutable references by directly casting the heap cell instead of wrapping the reference in a proxy. In this dissertation, an integration is proposed of monotonic references with the coercion-based solution to the problem of chains of proxies for other values such as functions. Furthermore, the prior semantics for monotonic references involved storing and evaluating cast expressions (not yet values) in the heap and it is not obvious how to implement this behavior efficiently in a compiler and run-time system. This dissertation proposes novel dynamic semantics where only values are written to the heap, making the semantics straightforward to implement. The approach is implemented in Grift, a compiler for a gradually typed programming language, and a few key optimizations are proposed. Finally, the proposed performance evaluation methodology shows that the proposed approach eliminates all overheads associated with gradually typed references in statically typed code regions without introducing significant average-case overhead.

Type Theories for Reactive Programming

CHRISTIAN ULDAL GRAULUND
IT University of Copenhagen, Denmark

Date: March 2021; Advisor: Rasmus Ejlers Møgelberg
URL: https://tinyurl.com/2fzwzea2

Functional reactive programming (FRP) is the application of techniques from functional programming to the domain of reactive programming. In recent years, there has been a growing interest in modal FRP. Here, modal types are added to languages for FRP with the goal of allowing the type system to enforce properties particular to reactive programming. These include causality, productivity and ruling out so-called space leaks.

The thesis is a collection of three previously published papers and an unpublished note. The first paper presents Simply RaTT, a simply typed language with with two Fitch-style modal type operators for reactive programming: one describing data available in the next time step, and one describing stable data that can be safely stored for the next time step. Recursion is introduced via a guarded fixed point operator. An operational semantics is given which allows for causal and productive evaluation of all stream transducers written in the language. Moreover, this evaluation is proved to be free of implicit space leaks. This paper was previously published at ICFP, 2019. The thesis also contains an unpublished note describing a denotational semantics for Simply RaTT in a presheaf category.

The second paper presents an extension of Lively RaTT, which allows one to also encode liveness properties such as fairness in the type system. Such properties are usually considered incompatible with guarded recursion, but the paper shows how these can be combined using a submodality of the step modality used in Simply RaTT. This paper was published at POPL 2021.

The final paper presents the language λ-Widget, a language designed for programming with widgets at the abstraction level of scene graphs, e.g., the DOM in a browser. This language uses a model of asynchronous events, designed for an efficient implementation strategy. The logical reading of λ-Widget combines linear temporal logic with linear logic. This paper was published at FoSSaCS 2021.

Deep and Shallow Types

BEN GREENMAN
Northeastern University, USA

Date: December 2020; Advisor: Matthias Felleisen
URL: https://tinyurl.com/n6a84ex7

The design space of mixed-typed languages is lively but disorganized. On one hand, researchers across academia and industry have contributed language designs that allow typed code to interoperate with untyped code. These design efforts explore a range of goals; some improve the expressiveness of a typed language, and others strengthen untyped code with a tailor-made type system. On the other hand, experience with type-sound designs has revealed major challenges. We do not know how to measure the performance costs of sound interaction. Nor do we have criteria that distinguish "truly sound" mixed-typed languages from others that enforce type obligations locally rather than globally.

In this dissertation, I introduce methods for assessing mixed-typed languages and bring order to the design space. My first contribution is a performance-analysis method that allows language implementors to systematically measure the cost of mixed-typed interaction.

My second contribution is a design-analysis method that allows language designers to understand implications of the type system. The method addresses two central questions: whether typed code can cope with untyped values, and whether untyped code can trust static types. Further distinctions arise by asking whether error outputs can direct a programmer to potentially-faulty interactions.

I apply the methods to several designs and discover limitations that motivate a synthesis of two ideas from the literature: deep types and shallow types. Deep types offer strong guarantees but impose a high interaction cost. Shallow types offer weak guarantees and better worst-case costs. This dissertation proves that deep and shallow types can interoperate and measures the benefits of a three-way mix.

Efficiency Three Ways: Tested, Verified, and Formalised

MARTIN ADAM THOMAS HANDLEY
University of Nottingham, UK

Date: September 2020; Advisor: Graham Hutton
URL: https://tinyurl.com/53959u36

Two fundamental goals in programming are correctness and efficiency: we want our programs to produce the right results, and to do so using as few resources as possible. One of the key benefits of the functional programming paradigm is the ability to reason about programs as if they are pure mathematical functions. In particular, programs can often be proved correct with respect to a specification by exploiting simple algebraic properties akin to secondary school mathematics. On the other hand, program efficiency is not immediately amenable to such algebraic methods used to explore program correctness. This insight manifests as a reasoning gap between program correctness and efficiency, and is a foundational problem in computer science. Furthermore, it is especially pronounced in lazy functional programming languages such as Haskell, where the on-demand nature of evaluation makes reasoning about efficiency even more challenging. To aid Haskell programmers in their reasoning about program efficiency, the work in this thesis seeks to partially bridge the reasoning gap using three different approaches: automated testing, semi-formal verification, and formal verification.

Formal Foundations for Provably Safe Web Components

MICHAEL HERZBERG
University of Sheffield, UK

Date: December 2019; Advisor: Achim D. Brucker
URL: https://tinyurl.com/tpusxam8

One of the cornerstones of modern software development that enables the creation of sophisticated software systems is the concept of reusable software components. Especially the fast-paced and business-driven web ecosystem is in need of a robust and safe way of reusing components. As it stands, however, the concepts and functions needed to create web components are spread out, immature, and not clearly defined, leaving much room for misunderstandings. To improve the situation, we need to look at the core of web browsers: the Document Object Model (DOM). It represents the state of a website with which users and client-side code (JavaScript) interact. Being in this central position makes the DOM the most central and critical part of a web browser with respect to safety and security, so we need to understand exactly what it does and which guarantees it provides. A well-established approach for this kind of highly critical system is to apply formal methods to mathematically prove certain properties. In this thesis, we provide a formal analysis of web components based on shadow roots, highlight their short-comings by proving them unsafe in many circumstances, and propose suggestions to provably improve their safety. In more detail, we build a formalisation of the Core DOM in Isabelle/HOL into which we introduce shadow roots. Then, we extract novel properties and invariants that improve the often implicit assumptions of the standard. We show that the model complies to the standard by symbolically evaluating all relevant test cases from the official compliance suite successfully on our model. We introduce novel definitions of web components and their safety and classify the most important DOM API accordingly, by which we uncover surprising behavior and shortcomings. Finally, we propose changes to the DOM standard by altering our model and proving that the safety of many DOM API methods improves while leading to a less ambiguous API.

Understanding and Evolving the Rust Programming Language

RALF JUNG
Saarland University, Germany

Date: August 2020; Advisor: Derek Dreyer
URL: https://tinyurl.com/38ejx5c3

Rust is a young systems programming language that aims to fill the gap between *high-level languages*—which provide strong static guarantees like memory and thread safety—and *low-level languages*—which give the programmer fine-grained control over data layout and memory management. This dissertation presents two projects establishing the first formal foundations for Rust, enabling us to better *understand* and *evolve* this important language: *RustBelt* and *Stacked Borrows*.

RustBelt is a formal model of Rust's type system, together with a soundness proof establishing memory and thread safety. The model is designed to verify the safety of a number of intricate APIs from the Rust standard library, despite the fact that the implementations of these APIs use *unsafe* language features.

Stacked Borrows is a proposed extension of the Rust specification, which enables the compiler to use the strong aliasing information in Rust's types to better analyze and optimize the code it is compiling. The adequacy of this specification is evaluated not only formally, but also by running real Rust code in an instrumented version of Rust's *Miri* interpreter that implements the Stacked Borrows semantics.

RustBelt is built on top of *Iris*, a language-agnostic framework, implemented in the Coq proof assistant, for building higher-order concurrent separation logics. This dissertation begins by giving an introduction to Iris, and explaining how Iris enables the derivation of complex high-level reasoning principles from a few simple ingredients. In RustBelt, this technique is exploited crucially to introduce the *lifetime logic*, which provides a novel separation-logic account of *borrowing*, a key distinguishing feature of the Rust type system.

Relational Reasoning for Effects and Handlers

CRAIG MCLAUGHLIN
University of Edinburgh, UK

Date: June 2020; Advisor: James McKinna and Ian Stark
URL: https://tinyurl.com/5cfp2ya9

This thesis studies relational reasoning techniques for *Frank*, a strict functional language supporting algebraic effects and their handlers, within a general, formalised approach for completely characterising observational equivalence.

Algebraic effects and handlers are an emerging paradigm for representing computational effects where primitive operations, the sources of an effect, are primary, and given semantics through their interpretation by effect handlers. Frank is a novel point in the design space because it recasts effect handling as part of a generalisation of call-by-value function application. Furthermore, Frank generalises unary effect handlers to the n-ary notion of *multihandlers*, supporting more elegant expression of certain handlers.

There have been recent efforts to develop sound reasoning principles, with respect to observational equivalence, for languages supporting effects and handlers. Such techniques support powerful equational reasoning about code, such as substitution of equivalent subterms ('equals for equals') in larger programs. However, few studies have considered a complete characterisation of observational equivalence, and its implications for reasoning techniques. Furthermore, there has been no account of reasoning principles for Frank programs.

Our first contribution is a formal reconstruction of a general proof technique, *triangulation*, for proving completeness results for observational equivalence. The technique brackets observational equivalence between two structural relations, a *logical* and an *applicative* notion. We demonstrate the triangulation proof method for a pure simply-typed λ-calculus. Our results are readily formalisable in *Agda* using state-of-the-art technology for dealing with syntaxes with binding.

Our second contribution is a calculus, *Ella*, capturing the essence of Frank's novel design. In particular, Ella supports binary handlers and generalises function application to incorporate effect handling. We extend our triangulation proof technique to this new setting, completely characterising observational equivalence for this calculus. We report on our partial progress in formalising our extension to Ella in Agda.

Our final contribution is the application of sound reasoning principles, inspired by existing literature, to a variety of Ella programs, including a proof of associativity for a canonical pipe multihandler. Moreover, we show how leveraging completeness leads, in certain instances, to simpler proofs of observational equivalence.

A Semantic Foundation for Sound Gradual Typing

MAX S. NEW
Northeastern University, USA

Date: December 2020; Advisor: Amal Ahmed
URL: https://tinyurl.com/y4jxarnn

Gradually typed programming languages provide a way forward in the debate between static and dynamic typing. In a gradual language, statically typed and dynamically typed programs can intermingle, and dynamically typed scripts can be gradually migrated to a statically typed style. In a *sound* gradually typed language, static type information is just as reliable as in a static language, establishing correctness of type-based refactoring and optimization. To ensure this in the presence of dynamic typing, runtime type casts are inserted automatically at the boundary between static and dynamic code. However the design of these languages is somewhat ad hoc, with little guidance on how to ensure that static reasoning principles are valid.

In my dissertation, I present a semantic framework for design and metatheoretic analysis of gradually typed languages based on the theory of embedding-projection pairs. I show that this semantics enables proofs of the fundamental soundness theorems of gradual typing, and that it is robust, applying it to different evaluation orders and programming features.

Accelerated Financial Algorithms: Derivative Pricing and Risk Management Applications

WOJCIECH MICHAL PAWLAK
University of Copenhagen, Denmark

Date: May 2021; Advisor: Martin Elsman, Cosmin Oancea, Allan Engsig-Karup and Carl Clausen
URL: https://tinyurl.com/3y5hwhvp

This industrial Ph.D. thesis is about Accelerated Financial Algorithms. We describe the design and implementation of common compute-intensive financial applications that combine state-of-the-art High-Performance Computing (HPC) code optimisation techniques to harness massive parallelism of modern parallel hardware architectures like Graphical Processing Units (GPU). We target the acceleration of pricing and risk management of real investment portfolios that consist of complex derivative instruments. We demonstrate our findings through a detailed analysis and practical accelerated implementations of common numerical algorithms. Our research is supplemented with a feasibility study carried out using high-level data-parallel programming languages and frameworks. We propose Futhark, which is a purely functional array programming language, as an example of a technology that enables efficient performance, while sustaining modularity, maintainability, and scalability of complex financial algorithms.

We focus on high-level algorithmic specifications and code optimisations that extract enough parallelism from a given algorithm to efficiently map it to high-throughput Graphics Processing Units (GPUs). In particular, we use flattening techniques for non-regular nested parallelism; target memory access patterns and size requirements through code optimisation such as data reordering, padding, and access coalescing; introduce inspector-executor approaches to dynamically analyse and adapt to any input dataset; and provide multiple kernel versions that together efficiently cover the entire spectrum of possible datasets.

The first contribution addresses an acceleration of a fixed-income derivatives pricing algorithm based on the Hull-White One-Factor Lattice Method (HW1F). We introduce a high-level algorithmic specification, which exhibits irregular-nested parallelism, and derive and optimise two hand tuned CUDA implementations: one of which utilises the only outer level of parallelism, while the other utilises both levels of parallelism. The second contribution is an accelerated algorithm for equity derivatives pricing that uses a Least Squares Monte Carlo Simulation Model (LSMC) following a Longstaff-Schwartz model and is implemented in high-level Futhark language. We show how an auto-generated implementation beats a manually-optimised one for certain datasets. The third contribution is a massive-scale Monte Carlo simulation to obtain Value at Risk (MCVaR) and other portfolio market risk measures. The memory-bound simulations comprise multiple nested parallel levels executed on a diverse portfolio of vanilla and exotic derivatives.

Leveraging Information Contained in Theory Presentations

YASMINE SHARODA
McMaster University, Canada

Date: March 2021 ; Advisor: Jacques Carette and William M. Farmer
URL: `https://tinyurl.com/4ma6p4j2`

Building a large library of mathematical knowledge is a complex and labour-intensive task. By examining current libraries of mathematics, we see that the human effort put in building them is not entirely directed towards tasks that need human creativity. Instead, a non-trivial amount of work is spent on providing definitions that could have been mechanically derived.

In this work, we propose a generative approach to library building, so definitions that can be automatically derived are computed by meta-programs. We focus our attention on libraries of algebraic structures, like monoids, groups, and rings. These structures are highly inter-related and their commonalities have been well-studied in universal algebra. We use theory presentation combinators to build a library of algebraic structures. Definitions from universal algebra and programming languages meta-theory are used to derive library definitions of constructions, like homomorphisms and term languages, from algebraic theory presentations. The result is an interpreter that, given 227 theory expressions, builds a library of over 5000 definitions. This library is, then, exported to Agda and Lean.

Don't Mind the Formalization Gap: The Design and Usage of hs-to-coq

ANTAL SPECTOR-ZABUSKY
University of Pennsylvania, USA

Date: May 2021; Advisor: Stephanie Weirich
URL: https://tinyurl.com/3e4cf36v

Using proof assistants to perform formal, mechanical software verification is a powerful technique for producing correct software. However, the verification is time-consuming and limited to software written in the language of the proof assistant. As an approach to mitigating this tradeoff, this dissertation presents hs-to-coq, a tool for translating programs written in the Haskell programming language into the Coq proof assistant, along with its applications and a general methodology for using it to verify programs. By introducing *edit files* containing programmatic descriptions of code transformations, we provide the ability to flexibly adapt our verification goals to exist anywhere on the spectrum between "increased confidence" and "full functional correctness".

Advanced Semantics for Non-deterministic and Probabilistic Programming

ALEXANDER VANDENBROUCKE
KU Leuven, Belgium

Date: December 2021; Advisor: Tom Schrijvers
URL: https://tinyurl.com/24mxekxz

Declarative programming is the idea that a program should describe *what* it does, *which* problem it is solving, not *how* it does so. Thus, declarative programming focuses on complexity that is inherent in the problem, not accidental complexity that originates from solving the problem on a computer. Less accidental complexity reduces the opportunity for software defects. This thesis contributes towards declarative programming in three overlapping paradigms: *logic programming*, *functional programming* and *probabilistic programming*.

The first part of this thesis studies tabling, a resolution mechanism for logic programming that ensures termination of a larger class of programs. Extensions of tabling (mode-directed tabling and answer subsumption) allow efficient tabling for optimisation problems. While much attention has been devoted to the expressivity and efficiency of these approaches, soundness has not been considered. This thesis shows that some implementations indeed fail to produce the correct answer for some programs. To remedy this situation, the thesis provides a formal framework and establishes a correctness criterion for soundness.

Then, tabling is adapted for non-determinism and optimisation in Haskell. However, Haskell's default recursion mechanism is not suitable for non-deterministic programs, and causes non-termination. Instead, the correct semantics is a complete-lattice based least fixed point semantics, implemented as a monad via tabling.

The second part of this thesis presents P$\lambda\omega$NK, a probabilistic programming language for modelling networks. Unlike the earlier language Probabilistic NetKAT (PNK), P$\lambda\omega$NK provides higher-order functions, to make programming more convenient and declarative.

Formalisation of P$\lambda\omega$NK is challenging for two reasons: Firstly, network programming introduces side effects (e.g., non-determinism and probabilistic choice) which need to be carefully controlled in a functional setting. P$\lambda\omega$NK's explicit syntax makes this interplay precise. Secondly, measure theory, the standard approach, only supports first-order functions. The solution is to leverage ω-Quasi Borel Spaces.

This work is not only useful for bringing abstraction to PNK. But may also inform similar meta-theoretic efforts which combine advanced features like higher-order functions, iteration and parallelism.