JFP **32**, e7, 11 pages, 2022. © The Author(s), 2022. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

doi:10.1017/S0956796822000053

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 ten 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

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Formally Verified Defensive Programming (Efficient Coq-Verified Computations From Untrusted ML Oracles)

SYLVAIN BOULMÉ
Université Grenoble Alpes, France

Date: September 2021; Advisor: None (Habilitation Thesis)
URL: `https://tinyurl.com/2dxvfznh`

We consider a lightweight approach —combining Coq and OCaml typecheckers— in order to formally verify higher-order imperative programs for partial correctness. In this approach, called FVDP (Formally Verified Defensive Programming), Coq-verified programs also contain some OCaml written functions, called oracles, which are untrusted: their implementation is simply ignored by the formal proof. Formal guarantees on the results of these oracles are obtained by combining dynamic tests in Coq with static properties deduced from OCaml types. Indeed, the simplest way to obtain a Coq-verified solver for a complex problem, often consists in combining an efficient OCaml oracle searching for "good" solutions with a verified defensive test able to ensure that the results found by the oracle have the expected properties. Hence, the solver benefits from the power of OCaml, while hiding many of its complex details to its formal proof: the theory of the defensive test is usually much simpler than the one of the oracle.

OCaml oracles are embedded into Coq through a "may-return monad", a structure that abstracts their side-effects by representing oracles as non-deterministic functions, and provided by my Impure library. This library also provides an elementary embedding of OCaml pointer equality, but powerful enough for defensive checks of some memoizing oracles (e.g. hash-consing of terms, memoized recursion). The thesis also shows how to deduce expressive invariants from polymorphic OCaml types, by adapting Wadler's "theorems for free". This technique is exploited within a design pattern—for certificate producing oracles—called Polymorphic LCF Style (or Polymorphic Factory Style). Large Coq proofs on these higher-order impure defensive computations are decomposed thanks to data-refinement techniques in order to cleanly separate reasoning on pure data-structures and algorithms from reasonings on sequences of impure computations. Then, the latter are (semi)automated thanks to computations of weakest liberal preconditions.

FVDP is detailed on several "realistic" applications: optimizing compilation (instruction scheduling for CompCert), static analysis (abstract domain of convex polyhedra of the VPL) and automated deduction (Boolean SAT-solving and linear rational arithmetic). The document explains how FVDP, instantiated with a careful software design, both alleviates development times and running times of such formally verified applications.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Efficient Implementations of Expressive Modelling Languages

GUERRIC CHUPIN
University of Nottingham, UK

Date: April 2022; Advisor: Henrik Nilsson
URL: https://tinyurl.com/bdzmypb7

This thesis is concerned with modelling languages aimed at assisting with modelling and simulation of systems described in terms of differential equations. These languages can be split into two classes: causal languages, where models are expressed using directed equations; and non-causal languages, where models are expressed using undirected equations.

This thesis focuses on two related paradigms: Functional Reactive Programming (FRP) and Functional Hybrid Modelling (FHM). FRP is an approach to programming causal time-aware applications that has successfully been used in causal modelling applications; while FHM is an approach to programming non-causal modelling applications. However, both are built on similar principles, namely, the treatment of models as first-class entities, allowing for models to be parameterised by other models or computed at runtime; and support for structurally dynamic models, whose behaviour can change during the simulation. This makes FRP and FHM particularly flexible and expressive approaches to modelling, especially compared to other mainstream languages. Because of their highly expressive and flexible nature, providing efficient implementations of these languages is a challenge. This thesis explores novel implementation techniques aimed at improving the performance of existing implementations of FRP and FHM, and other expressive modelling languages built on similar ideas.

In the setting of FRP, this thesis proposes a novel embedded FRP library that uses the implementation approach of synchronous dataflow languages. This allows for significant performance improvement by better handling of the reactive network's topology, which represents a large portion of the runtime in current implementations, especially for applications that make heavy use of continuously varying values, such as modelling applications.

In the setting of FHM, this thesis presents the modular compilation of a language based on FHM. Due to inherent difficulties with the simulation of systems of undirected equations, previous implementations of FHM and similarly expressive languages were either interpreted or generated code on the fly using just-in-time compilation, two techniques which have runtime overhead over ahead-of-time compilation. This thesis presents a new method for generating code for equation systems which allows for the separate compilation of FHM models.

Compared with current approaches to FRP and FHM implementation, there is greater commonality between the implementation approaches described here, suggesting a possible way forward towards a future non-causal modelling language supporting FRP-like features, resulting in an even more expressive modelling language.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Pattern-Matching-Oriented Programming Language and Computer Algebra System as Its Application

SATOSHI EGI
University of Tokyo, Japan

Date: March 2022; Advisor: Masami Hagiya
URL: https://tinyurl.com/2p8baer5

A new programming language that widens the range of algorithms that we can concisely describe is important not only because it makes programming for the existing problems easy but also because it widens the scope of computer science by enabling us to deal with problems that we have avoided because programming has been difficult. In this thesis, we propose the Egison programming language with two independent new features: (i) a pattern-match facility for non-free data types; (ii) a facility for describing tensor calculus in differential geometry in a form similar to mathematical formulae using index notation.

Pattern matching of Egison features user-customizable non-linear pattern matching with backtracking. In the first half of the thesis, we discuss the design and implementation of this pattern-match facility, classify programming techniques utilizing this pattern-match facility, and advocate a new paradigm, called pattern-match-oriented programming. Pattern-match-oriented programming simplifies definitions of many recursive programs by confining explicit recursions for backtracking inside an intuitive pattern. Egison pattern matching has two implementations: an interpreter in Haskell and a library implemented using Haskell meta-programming. A computer algebra system that supports tensor index notation is implemented in the Egison interpreter using this Haskell library.

In the second half of the thesis, we propose a method for importing tensor index notation into programming. First, we propose a set of symbolic index reduction rules that allow us to concisely define tensor operators, such as tensor addition and multiplication, and partial derivative. Second, we propose a set of index completion rules that allow us to concisely define operators for differential forms, such as wedge product, exterior derivative, and Hodge operator. Our method allows the users to write a program in a form close to formulae in differential geometry. Therefore, our system is easy to use even for researchers of mathematics who are not used to programming.

Finally, we discuss a general method for creating a new language facility by reviewing the processes of designing these two language facilities. To this end, we divide the process of creating a new language facility into three steps, classify our contributions into these three steps, and list commonalities.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Tangible Values With Text: Explorations of Bimodal Programming

BRIAN HEMPEL
University of Chicago, USA

Date: March 2022; Advisor: Ravi Chugh
URL: `https://tinyurl.com/5bx7mpxv`

Direct manipulation is everywhere. While the intuitive "point-click-operate" workflow of direct manipulation is the standard mode of interaction for most computer applications, for over half a century one important application has remained a text-based activity: programming. Can the intuitive workflow of direct manipulation be applied to programming—could programming become as simple as manipulating the program's output, showing the computer what you want it to do? Alas, 45 years of research on this "programming by demonstration" (PBD) vision has yielded only niche successes.

To confront this impasse, this dissertation reverses a key assumption of PBD systems. Traditional PBD systems eschew textual code, assuming that textual code is difficult for users. But, whatever its faults, textual code is a proven paradigm for understanding and editing programs. Therefore, this work instead embraces textual code: we start with text-based programming in a generic programming language and, rather than replace text, augment it with PBD-style direct manipulation on visualized program outputs. Output manipulations induce changes to the textual code. Such a system is bimodal: at any time, users may program via text edits on code or via mouse manipulations on outputs.

To explore the expressiveness of this bimodal approach, this work presents two programming systems. The first system, called Sketch-n-Sketch, mimics a traditional graphics editor, enabling users to use standard drawing interactions to create programs that output vector graphics. The second, called Maniposynth, brings output-based interaction closer to ordinary programming, offering a graphical interface for constructing OCaml programs that operate on functional data structures. We show the expressive extent of direct manipulation in both systems through examples. Overall, this work expands and illuminates the capabilities of bimodal programming.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Foundations for Programming and Implementing Effect Handlers

DANIEL HILLERSTRÖM
University of Edinburgh, UK

Date: March 2022; Advisor: Sam Lindley and John Longley
URL: https://tinyurl.com/4vbwekc5

Effect handler oriented programming (EHOP) is a paradigm in which programs are syntax whose semantics are compartmentalised into a collection of effect handlers. The separation of syntax and semantics provides a modular basis for building software, where programs can be retrofitted with additional functionality in a backwards compatible way. My dissertation comprises three strands of work on EHOP.

The first strand demonstrates EHOP by example. The example is a tiny UNIX-style operating system with functionality such as multiple user environment/sessions, process multi-tasking and interruption, a file system, a programmable shell environment, etc. This system is built iteratively, starting from a very basic notion of input/output, and, then subsequently extended with more functionality by seamlessly composing ever more effect handlers.

The second strand develops two canonical implementation techniques for effect handlers. The first technique is a continuation passing style transform based on the novel notion of generalised continuations. A generalised continuation is a high-level abstraction, which captures the essence of the low-level runtime stack manipulations that occur in native implementations such as segmented stacks. The second technique is a CEK-style abstract machine, which is readily obtained by swapping out the K-component of the original CEK machine with a generalised continuation. This machine provides a well-understood operational foundation for implementing effect handlers.

The third strand explores the expressive power of effect handlers. This strand consists of two results. The first result shows that the deep, shallow, and parameterised flavours of effect handlers are typability-preserving macro-expressible. The second result shows that effect handlers enable some programs to run asymptotically faster. To establish this result I use the problem of 'generic count' and the notion of type-respecting expressiveness. Specifically, I show that every possible implementation of generic count in a language without effect handlers can at best have $\Omega(n2^n)$ runtime, where n is the size of the problem, whilst by extending said language with effect handlers there exist implementations that admit $\mathcal{O}(2^n)$ runtimes, an asymptotic gain of a factor of n.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

A Semantic Foundation for Gradual Set-Theoretic Types

VICTOR LANVIN
Université de Paris, France

Date: November 2021; Advisor: Giuseppe Castagna
URL: `https://tinyurl.com/23pbvfzx`

In this thesis, we study the interaction between set-theoretic types and gradual typing. Set-theoretic types are well-suited to a semantic-based approach called "semantic subtyping", in which types are interpreted as sets of values, and subtyping is defined as set-containment between these sets. We adopt this approach throughout the entirety of this thesis. Since set-theoretic types are characterized by their semantic properties, they can be easily embedded in existing type systems. This contrasts with gradual typing, which is an intrinsically syntactic concept since it relies on the addition of a type annotation to inform the type-checker not to perform some checks.

In this thesis, we try and reconcile the two concepts, by proposing several semantic interpretations of gradual typing. In the first part, we propose a new approach to integrate gradual typing in an existing static type system. The originality of this approach comes from the fact that gradual typing is added in a declarative way to the system by adding a single logical rule. As such, we do not need to revisit and modify all the existing rules. While this first part of the thesis can be seen as a logical approach to tackle this problem, the second part sets off along a more semantic strategy. In particular, we study whether it is possible to reconcile the interpretation of types proposed by the semantic subtyping approach and the interpretation of the terms of a language. The ultimate goal being to define a denotational semantics for a gradually-typed language. We tackle this problem in several steps. First, we define a denotational semantics for a simple lambda-calculus with set-theoretic types, based directly on the semantic subtyping approach. We then extend this by giving a formal denotational semantics for the functional core of CDuce, a language featuring set-theoretic types and several complex constructs, such as type-cases, overloaded functions, and non-determinism. Finally, we study a gradually-typed lambda-calculus, for which we present a denotational semantics. We also give a set-theoretic interpretation of gradual types, which allows us to derive some very powerful results about the representation of gradual types.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Optimized and Formally Verified Compilation for a VLIW Processor

CYRIL SIX
Université Grenoble Alpes, France

Date: July 2021; Advisor: David Monniaux, Sylvain Boulmé and Benoît Dupont de Dinechin
URL: https://tinyurl.com/3wbv89j

CompCert is a success story of functional programming in Coq and OCaml. It is the first optimizing compiler — used in industry — with a formal proof of correctness: compiled programs are proven to behave the same as their source programs. However, because of the challenges involved in proving compiler optimizations, CompCert only has a limited number of them. While this may not significantly impact out-of-order architectures such as x86, on in-order architectures, particularly on VLIW processors, CompCert usually generates low-performance code compared to classical compilers such as GCC (code running half as fast as GCC -O2). On VLIW processors, the intra-level parallelism is explicit and must be specified in the assembly code through "bundles" of instructions: the compiler must bundlize instructions to achieve good performance.

In this thesis, we identify, investigate, implement and formally verify several classical optimizations missing in CompCert. We start by introducing a formal model for VLIW bundles execution on an interlocked core and generate those bundles through a postpass — after register allocation — scheduling . Then, we introduce a prepass — before register allocation — superblock scheduling, implementing static branch prediction and tail-duplication along the way. Finally, we further increase the performance of our generated code by implementing loop unrolling, loop rotation and loop peeling—the latter being used for Loop-Invariant Code Motion. These transformations are verified by translation validation, some of them with hash-consing to achieve reasonable compilation time. We evaluate each introduced optimization on benchmarks, including Polybench and TACleBench, on the KV3 VLIW core, ARM Cortex A53, and RiscV “Rocket” core. Thanks to this work, our version of CompCert is now only 16% slower (respectively 12% slower and 30% slower) than GCC -O2 on the KV3 (respectively ARM and RiscV), instead of 50% (respectively 38% and 45%).

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory

JONATHAN STERLING
Carnegie Mellon University, USA

Date: October 2021; Advisor: Robert Harper
URL: https://tinyurl.com/2p8tw6pv

The implementation and semantics of dependent type theories can be studied in a syntax-independent way: the objective metatheory of dependent type theories exploits the universal properties of their syntactic categories to endow them with computational content, mathematical meaning, and practical implementation (normalization, type checking, elaboration). The semantic methods of the objective metatheory inform the design and implementation of correct-by-construction elaboration algorithms, promising a principled interface between real proof assistants and ideal mathematics.

In this dissertation, I add synthetic Tait computability to the arsenal of the objective metatheorist. Synthetic Tait computability is a mathematical machine to reduce difficult problems of type theory and programming languages to trivial theorems of topos theory. First employed by Sterling and Harper to reconstruct the theory of program modules and their phase separated parametricity, synthetic Tait computability is deployed here to resolve the last major open question in the syntactic metatheory of cubical type theory: normalization of open terms.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

A Language-based Approach to Programming with Serialized Data

MICHAEL VOLLMER
Indiana University, USA

Date: February 2021; Advisor: Ryan Newton
URL: `https://tinyurl.com/mr8m2648`

In a typical data-processing application, the representation of data in memory is distinct from its representation in a serialized form on disk. The former has pointers and an arbitrary, sparse layout, facilitating easier manipulation by a program, while the latter is packed contiguously, facilitating easier I/O. I propose a programming language, LoCal, that unifies the in-memory and on-disk representations of data. LoCal extends prior work on region calculi into a location calculus, employing a type system that tracks the byte-addressed layout of all heap values. I present the formal semantics of LoCal and prove type safety, and show how to infer LoCal programs from unannotated source terms. Then, I demonstrate how to efficiently implement LoCal in a practical compiler that produces code competitive with hand-written C.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053

Executable Examples: Empowering Students to Hone Their Problem Comprehension

JOHN WRENN
Brown University, USA

Date: May 2022; Advisor: Shriram Krishnamurthi
URL: `https://tinyurl.com/my8bm5ks`

Students often tackle programming problems with a flawed understanding of what the problem is asking. Some pedagogies attempt to address this by encouraging students to develop examples in the form of input–output assertions (henceforth "functional examples"), independent of (and typically prior to) developing and testing their implementations. However, without an implementation to run examples against, examples are impotent and do not provide feedback. Consequently, students may be inclined to begin their implementations prematurely–a process whose comparatively ample feedback may mask underlying misunderstandings and instill a false sense of progress.

In this dissertation, I demonstrate that providing students with timely feedback on their functional examples incentivizes them to develop functional examples, improves the quality of their test cases, and may improve the correctness of their implementations.

Downloaded from https://www.cambridge.org/core. IP address: 18.226.88.63, on 21 Feb 2025 at 15:52:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796822000053