JFP **31**, e1, 13 pages, 2021.
doi:10.1017/S0956796820000301

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 twelve 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

Design and Implementation of Effect Handlers for Object-Oriented Programming Languages

JONATHAN IMMANUEL BRACHTHÄUSER
Eberhard Karls Universität Tübingen, Germany

Date: May 2020; Advisor: Klaus Ostermann
URL: https://tinyurl.com/yxvgtk5j

Algebraic effects and their extension with handlers offer interesting new ways to structure programs. Effect handlers support two important aspects of software development concisely: they can express advanced control-flow structures as well as facilitate parametrization of software components. Unifying both aspects also guarantees well-defined interaction between control flow and parametrization. Despite their recently growing popularity, we identify two problems hindering adoption by a wider audience of programmers. Firstly, programmers are immediately confronted with the full generality of effect handlers. While effect handlers are expressive enough to model advanced control-flow structures, not all use cases require this expressivity. Secondly, effect handlers have been conceived in the realm of functional programming languages and have almost exclusively been studied in the context of functional programming.

In this thesis, we propose solutions to the two aforementioned problems with the goal to facilitate adoption of effect handlers by a wider audience.

To address the first problem, we systematically present effect handlers as a combination of delimited control (that is, they allow control-flow transfers) and dynamic binding (that is, they allow parametrization). We discover that dynamic binding and effect handlers form a spectrum: a novel intermediate form of *ambient functions* enables abstraction similar to effect handlers, but without modifying the control flow: Ambient functions are dynamically bound, but statically evaluated. Introducing effect handlers from dynamic binding offers programmers an alternative way to approach handlers. They can incrementally learn and understand the different generalizations.

To address the second problem, we present a design to embed effect handlers in object-oriented programming languages. Our design embraces the object-oriented programming paradigm and we map abstractions of effect handlers to key abstractions of object-oriented programming. Combining the two paradigms not only enables programmers to use effect handlers in object-oriented programs, but also to use object-oriented programming abstractions to modularize effect handlers. Our design employs *explicit capability-passing style*. That is, instead of dynamically searching for a handler at runtime, we pass instances of handlers as additional arguments to methods. We present multiple implementations of our design and study the extensibility properties gained by embedding effect handlers into object-oriented programming languages.

A Constructive Calculus for Esterel

SPENCER P. FLORENCE
Northwestern University, USA

Date: September 2020; Advisor: Robert Bruce Findler
URL: https://tinyurl.com/y2nqncr7

The language Esterel has found success in many safety-critical applications, from aircraft landing gear to digital signal processors. Its unique combination of powerful control operations, deterministic concurrency, and real time execution bounds are indispensable to programmers in these kinds of safety-critical domains. However these features lead to an interesting facet of the language, called Constructivity.

Constructivity is a non-local property of Esterel programs which makes defining semantics for the language subtle. Existing semantics tend to sacrifice some desirable facet of a language semantics to handle this. Many sacrifice locality, and only work on whole programs. Some sacrifice adequacy, allowing them to describe transformations to programs at the cost of being able to actually run programs. Still more decide to work in a domain other than Esterel, such as circuits, making Constructivity easier to capture, but forcing users of these semantics to reason in a domain which they are not programming in.

This dissertation provides the first semantics for Esterel which captures all of the above facets, while still describing Constructivity.

https://doi.org/10.1017/S0956796820000301 Published online by Cambridge University Press

Mechanized Verification of the Correctness and Asymptotic Complexity of Programs: The Right Answer at the Right Time

ARMAËL GUÉNEAU
Université de Paris, France

Date: December 2019; Advisor: François Pottier and Arthur Charguéraud
URL: https://tinyurl.com/y3t68osm

This dissertation is concerned with the question of formally verifying that the implementation of an algorithm is not only functionally correct (it always returns the right result), but also has the right asymptotic complexity (it reliably computes the result in the expected amount of time).

In the algorithms literature, it is standard practice to characterize the performance of an algorithm by indicating its asymptotic time complexity, using Landau's "big-O" notation. We argue that asymptotic complexity bounds are equally useful as formal specifications, because they enable modular reasoning: they abstract over the concrete cost expression of a program, and therefore abstract over the specifics of its implementation. We describe a number of challenges with the use of the O notation, especially in the multivariate case, that might be overlooked when reasoning informally.

We put these considerations into practice by formalizing the O notation in the Coq proof assistant, and by extending an existing program verification framework with a methodology for establishing robust and modular proofs of asymptotic complexity bounds. We extend Separation Logic with Time Credits, which allows reasoning at the same time about correctness and time complexity, and introduce negative time credits. Negative time credits increase the expressiveness of the logic, and enable convenient reasoning principles as well as elegant specifications. To establish such specifications, we develop a methodology that allows proofs of complexity in Separation Logic to be robust and carried out at a relatively high level of abstraction, based on mechanisms for: collecting and deferring constraints during the proof, and semi-automatically synthesizing cost expressions without loss of generality.

We demonstrate the usefulness and practicality of our approach on a number of increasingly challenging case studies. We start with simple algorithms and data structures, and ramp up to our most challenging case study: the proof of correctness and amortized complexity of a state-of-the-art incremental cycle detection algorithm. Thus, our methodology scales up to highly non-trivial algorithms whose complexity analysis depends on subtle functional invariants, and can formally OCaml libraries which are then actually usable as part of real world programs.

Operational Semantics of Weak Sequential Composition

HENDRIK MAARAND
Tallinn University of Technology, Estonia

Date: June 2020; Advisor: Tarmo Uustalu
URL: https://tinyurl.com/y5asuw5a

This dissertation proposes an operational semantics where sequential composition can be relaxed for certain pairs of instructions. By this we mean that instructions are not necessarily executed in the order given by the program. Our motivation is that programs are often executed in such a relaxed manner: it is not always guaranteed that the effect of an instruction earlier in the program becomes visible before the effect of an instruction later in the program. For example, the hardware may execute instructions out-of-order. A typical requirement is that such relaxations should not be visible on sequential programs. Even then, such relaxations may become visible on concurrent programs. Formal description of step-by-step execution of programs under such relaxed sequential composition provides a foundation for trustworthy analysis of programs.

In our approach, we consider the set of (primitive) instructions as an alphabet and represent programs as regular expressions over this alphabet. Program executions are words over this alphabet. The pairs of instructions for which sequential composition is weak is given by an independence relation on the alphabet (as in Mazurkiewicz traces). Given a program, the operational semantics should tell us which instruction we can execute next and what is the residual program after that. As a first step towards the desired operational semantics, we generalise the Brzozowski and Antimirov syntactic derivative operations to what we call reordering derivatives. These allow us to construct letter-by-letter any word in the trace closure of the language of the regular expression. The basic operational semantics is obtained from the Antimirov reordering derivative by adding parallel composition to the syntax, adding machine states to the rules and interpreting the letters of the alphabet as state transformers. We then also consider some extensions to describe more intricate behaviours. As an experiment, we describe a fragment of the multicopy-atomic ARMv8 memory model in this framework. Unrelated to relaxed memory concurrency, we also investigate when is the set of reordering derivatives of a regular expression finite, i.e., when is the trace closure of the language of a regular expression regular.

Reasoning About Effectful Programs and Evaluation Order

DYLAN MCDERMOTT
University of Cambridge, UK

Date: October 2019; Advisor: Alan Mycroft
URL: https://tinyurl.com/yyvqwepn

Program transformations have various applications, such as in compiler optimizations. These transformations are often effect-dependent: replacing one program with another relies on some restriction on the side-effects of subprograms. For example, we cannot eliminate a dead computation that raises an exception, or a duplicated computation that prints to the screen. Effect-dependent program transformations can be described formally using effect systems, which annotate types with information about the side-effects of expressions.

In this thesis, we extend previous work on effect systems and correctness of effect-dependent transformations in two related directions.

First, we consider evaluation order. Effect systems for call-by-value languages are well-known, but are not sound for other evaluation orders. We describe sound and precise effect systems for various evaluation orders, including call-by-name. We also describe an effect system for Levy's call-by-push-value, and show that this subsumes those for call-by-value and call-by-name. This naturally leads us to consider effect-dependent transformations that replace one evaluation order with another. We show how to use the call-by-push-value effect system to prove the correctness of transformations that replace call-by-value with call-by-name, using an argument based on logical relations. Finally, we extend call-by-push-value to additionally capture call-by-need. We use our extension to show a classic example of a relationship between evaluation orders: if the side-effects are restricted to (at most) nontermination, then call-by-name is equivalent to call-by-need.

The second direction we consider is non-invertible transformations. A program transformation is non-invertible if only one direction is correct. Such transformations arise, for example, when considering undefined behaviour, nondeterminism, or concurrency. We present a general framework for verifying noninvertible effect-dependent transformations, based on our effect system for call-by-push-value. The framework includes a non-symmetric notion of correctness for effect-dependent transformations, and a denotational semantics based on order-enriched category theory that can be used to prove correctness.

Type-Safe Generic Differencing of Mutually Recursive Families

VICTOR CACCIARI MIRALDO
Utrecht University, the Netherlands

Date: October 2020; Advisor: Gabriele Keller and Wouter Swierstra
URL: https://tinyurl.com/y5bs5k4f

The UNIX diff tool – which computes the differences between two files in terms of a set of copied lines – is widely used in software version control. The fixed *lines-of-code* granularity, however, is sometimes too coarse and obscures simple changes, i.e., renaming a single parameter triggers the whole line to be seen as *changed*. This may lead to unnecessary conflicts when unrelated changes occur on the same line. Consequently, it is difficult to merge such changes automatically.

In this thesis we discuss two novel approaches to structural differencing, generically – which work over a large class of datatypes. The first approach defines a type-indexed representation of patches and provides a clear merging algorithm, but it is computationally expensive to produce patches with this approach. The second approach addresses the efficiency problem by choosing an extensional representation for patches. This enables us to represent transformations involving insertions, deletions, duplication, contractions and permutations which are computable in linear time. With the added expressivity, however, comes added complexity. Consequently, the merging algorithm is more intricate and the patches can be harder to reason about.

Both of our approaches can be instantiated to mutually recursive datatypes and, consequently, can be used to compare elements of most programming languages. Writing the software that does so, however, comes with additional challenges. To address this we have developed two new libraries for generic programming in Haskell.

Finally, we empirically evaluate our algorithms by a number of experiments over real conflicts gathered from GitHub. Our evaluation reveals that at least 26% of the conflicts that developers face on a day-to-day basis could have been automatically merged. This suggests there is a benefit in using structural differencing tools as the basis for software version control.

Assisting End Users in Workflow Systems

NICO NAUS
Utrecht University, The Netherlands

Date: June 2020; Advisor: Johan Jeuring
URL: https://tinyurl.com/yyaywdal

In today's society, almost every company and institution employs some kind of workflow automation. Hospitals employ software that automates health care processes. The coastal guard uses workflow software to assist in search and rescue operations. Naval ships use workflow automation software to manage people, resources and mission goals.

Before automation, users knew the process by heart, and knew how their choices influenced the process. Workflow systems hide the flow of processes behind interfaces. For end users, it is not always clear how decisions influence the progress of a task. One way to provide users with more information about their current situation is to provide them with next-step hints. These hints are based on their current situation: their position in the workflow and the data in the system. In this dissertation, I attempt to answer the question, how can we provide end users with next-step hints to aid them in making decisions?

The answer to that question is found by applying of techniques from intelligent tutoring systems (ITS) and program analysis. Previous work on ITS strategies inspired the first approach to generate next-step hints. By extending the original program with additional information, it can be viewed as a rule-based problem, making it susceptible to generic AI search and solving algorithms. The second approach comes from program analysis. By employing symbolic execution, next-step hints are automatically calculated, without any changes to the original code.

The application of both techniques results in two next-step hints systems. One system, aided by the programmer, the other fully automatic. In developing the automatic system, a formal task-oriented programming semantics is also developed, including a symbolic execution semantics. Both systems are proven to be sound and complete. They are both implemented, too, showing that they work in practice. Providing next-step hints to end users is crucial in improving the quality of decisions. It helps end users by giving insight into the effects of their choices, and makes sure that all data is taken into consideration.

Contributions to Multimode and Presheaf Type Theory

ANDREAS NUYTS
KU Leuven, Belgium

Date: August 2020; Advisor: Frank Piessens and Dominique Devriese
URL: https://tinyurl.com/yxwvxnld

Dependent type theory is a theoretical programming language in which programmers can not only write their programs but also prove properties of these programs (e.g. that they satisfy their specification) and have these proofs checked by the type-checker. This type-checker is however a bad understander and requires a proof of every lemma invoked, no matter how obvious. This damages the practical usability of these languages.

Reynolds' framework of relational parametricity, which is based on the observation that all type formers in a well-behaved language have an action on relations and that all polymorphic functions in such language respect these relations (they are parametric), gives us a large class of "obvious" theorems for free. Based on the observation that all functions in a well-behaved language respect isomorphism, homotopy type theory (HoTT) gives us a different class of "obvious" theorems and isomorphisms for free.

Neither framework is entirely satisfying. Indeed, while all functions have a graph relation, a relation in general cannot be applied to an input to compute an output, so that preservation of relations is often insufficient. On the other hand, while all isomorphisms are functions, the converse does not hold.

Directed type theory should start from the observation that all covariant type formers have an action on transformations, and that all polymorphic functions between such types commute with these transformations (they are natural). Thus, we expect that directed type theory can give us a class – larger than the aforementioned ones – of theorems and computable transformations for free.

Sadly, not all type formers are covariant and not all polymorphic functions of interest are necessarily parametric or natural. Thus, we need an (ideally automatic) bookkeeping system for keeping track of which functions are and which are not. This bookkeeping system is provided by modal (and more generally multimode) type theory, where every function is annotated by a modality describing its behaviour.

This thesis makes several contributions in the areas of modal (multimode) and presheaf type theory, which have several applications but are motivated by the development of a directed type theory that attains the aforementioned goals.

On the Implementation of Purely Functional Data Structures for the Linearisation case of Dynamic Trees

JUAN CARLOS SAENZ CARRASCO
University of Sheffield, UK

Date: August 2020; Advisor: Mike Stannet
URL: https://tinyurl.com/y4lwb3nq

Dynamic trees, originally described by Sleator and Tarjan, have been studied in detail for non persistent structures providing $O(\log n)$ time for update and lookup operations as shown in theory and practice by Werneck. However, there are two gaps in current theory. First, how the most common dynamic tree operations (link and cut) are computed over a purely functional data structure has not been studied in detail. Second, even in the imperative case, when checking whether two vertices u and v are connected (i.e. in the same component), it is taken for granted that the corresponding location indices (i.e. pointers, which are not allowed in purely functional programming) are known a priori and do not need to be computed, yet this is rarely the case in practice. In this thesis we address these omissions by formally introducing two new data structures, Full and Top, which we use to represent trees in a functionally efficient manner. Based on a primitive version of finger trees – the de facto sequence data structure for the purely lazy-evaluation programming language Haskell – they are augmented with collection (i.e. set-based) data structures in order to manage efficiently k-ary trees for the so-called linearisation case of the dynamic trees problem. Different implementations are discussed, and their performance is measured. Our results suggest that relative timings for our proposed structures perform sublinear time per operation once the forest is generated. Furthermore, Full and Top implementations show simplicity and preserve purity under a common interface.

Higher Inductive Types, Inductive Families, and Inductive-Inductive Types

JAKOB VON RAUMER
University of Nottingham, UK

Date: February 2020; Advisor: Thorsten Altenkirch
URL: https://tinyurl.com/y2mlslj7

Martin-Löf type theory is a formal language which is used both as a foundation for mathematics and the theoretical basis of a range of functional programming languages. Inductive types are an important part of type theory which is necessary to express data types by giving a list of rules stating how to form this data. In this thesis we we tackle several questions about different classes of inductive types.

In the setting of homotopy type theory, we will take a look at higher inductive types based on homotopy coequalizers and characterize their path spaces with a recursive rule which looks like an induction principle. This encapsulates a proof technique known as "encode-decode method".

In an extensional meta-theory we then explore the phenomenon of induction-induction, specify inductice families and discuss how we can reduce each instance of an inductive-inductive type to an inductive family. Our result suggests a way to show that each type theory which encompasses inductive families can also express all inductive-inductive types.

Equality Between Programs With Effects

NIELS VOORNEVELD
University of Ljubljana, Slovenia

Date: January 2020; Advisor: Alex Simpson
URL: https://tinyurl.com/y2zcegvr

This thesis studies notions of program equivalence for a call-by-push-value functional language with algebraic effects and general recursion. We mainly focus on behavioural equivalence, where program behaviour is specified by a collection of effect-specific formulas. Two programs of the same type are deemed equivalent if they satisfy the same formulas. To interpret effectful behaviour in a generic way, computation terms are evaluated to trees built from effect operators. These trees are then interpreted in a logic using modalities, which lift predicates on value types to predicates on computation types.

One of the main contributions of this thesis is identifying conditions on the modalities under which the behavioural equivalence induced by the logic is a congruence. This means equivalent terms cannot be distinguished by programs. To prove this property, we show that the behavioural equivalence coincides with an appropriate notion of applicative bisimilarity, where effects are interpreted using relators (which lift relations). This allows us to prove the aforementioned congruence using a variation of Howe's method.

The algebraic effects to which the results apply include error, nondeterminism, probability, global store, input/output, and timer. Several combinations of these effects can also be described with the logic. However, in order to combine effects more easily, and to give more natural descriptions of program behaviour, the logic is generalised to a logic with quantitative formulas. Once again, the congruence property and connections with applicative bisimilarity are established.

Finally, we show that similar results hold also if the language is extended with additional type constructors. In particular, we consider universal polymorphic and recursive types.

Debugging Functional Programs by Interpretation

JOHN WHITINGTON
University of Leicester, UK

Date: July 2020; Advisor: Tom Ridge
URL: https://tinyurl.com/yyg648mv

Motivated by experience in programming and in the teaching of programming, we make another assault on the longstanding problem of debugging. Having explored why debuggers are not used as widely as one might expect, especially in functional programming environments, we define the characteristics of a debugger which make it usable and thus likely to be widely used. We present work on a new debugger for the functional programming language OCaml which operates by direct interpretation of the program source, allowing the printing out of individual steps of the program's evaluation, and discuss its technical implementation and practical use. It has two parts: a stand-alone debugger which can run OCaml programs by interpretation and so allow their behaviour to be inspected; and an OCaml syntax extension, which allows the part of a program under scrutiny to be interpreted in the same fashion as the stand-alone debugger whilst the rest of the program runs natively. We show how this latter mechanism can create a source-level debugging system that has the characteristics of a usable debugger and so may eventually be expected to be suitable for widespread adoption.

https://doi.org/10.1017/S0956796820000301 Published online by Cambridge University Press