JFP **29**, e1, 10 pages, 2019.
doi:10.1017/S0956796818000230

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 nine 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

Functional Abstraction for Programming Multi-Level Architectures: Formalisation and Implementation

VICTOR ALLOMBERT
Université Paris-Est, France

Date: July 2017; Advisor: Frédéric Gava and Julien Tesson
URL: https://tinyurl.com/ybvp8b5x

From personal computers using an increasing number of cores, to supercomputers having millions of computing units, parallel architectures are the current standard. The high performance architectures are usually referenced to as hierarchical, as they are composed from clusters of multi-processors of multi-cores. Programming such architectures is known to be notoriously difficult. Writing parallel programs is, most of the time, difficult for both the algorithmic and the implementation phase. To answer those concerns, many structured models and languages were proposed in order to increase both expressiveness and efficiency. Among other models, Multi-BSP is a bridging model dedicated to hierarchical architecture that ensures efficiency, execution safety, scalability and cost prediction. It is an extension of the well known BSP model that handles flat architectures.

In this thesis we introduce the Multi-ML language, which allows programming Multi-BSP algorithms "à la ML" and thus, guarantees the properties of the Multi-BSP model and the execution safety, thanks to a ML type system. To deal with the multi-level execution model of Multi-ML, we defined formal semantics which describe the valid evaluation of an expression. To ensure the execution safety of Multi-ML programs, we also propose a typing system that preserves replicated coherence. An abstract machine is defined to formally describe the evaluation of a Multi-ML program on a Multi-BSP architecture. An implementation of the language is available as a compilation toolchain. It is thus possible to generate an efficient parallel code from a program written in Multi-ML and execute it on any hierarchical machine.

Adventures in Formalisation: Financial Contracts, Modules, and Two-Level Type Theory

DANIL ANNENKOV
University of Copenhagen, Denmark

Date: May 2018; Advisor: Martin Elsman
URL: https://tinyurl.com/ycf4gabn

This thesis present three projects concerned with applications of certified programming techniques and proof assistants in the area of programming language theory and mathematics.

The first project develops a certified compilation technique for a domain-specific programming language for financial contracts (the CL language). The code in CL is translated into a simple expression language well-suited for integration with software components implementing Monte Carlo simulation techniques (pricing engines). The compilation procedure is accompanied with formal proofs of correctness carried out in the Coq proof assistant. Moreover, we develop techniques for capturing the dynamic behaviour of contracts with the passage of time. These techniques potentially allow for efficient integration of contract specifications with high-performance pricing engines running on GPGPU hardware.

The second project presents a number of techniques that allow for formal reasoning with nested and mutually inductive structures built up from finite maps and sets (also called semantic objects). The techniques, which build on the theory of nominal sets combined with the ability to work with multiple isomorphic representations of finite maps, make it possible to give a formal treatment, in Coq, of a higher-order module system, including the ability to eliminate entirely, at compile time, abstraction barriers introduced by the module system. The development is based on earlier work on static interpretation of modules and provides the foundation for a higher-order module language for Futhark, an optimising compiler targeting data-parallel architectures, such as GPGPUs.

The third project presents an implementation of two-level type theory, a version of Martin-Lof type theory with two equality types: the first acts as the usual equality of homotopy type theory, while the second allows us to reason about strict equality. In this system, we can formalise results of partially meta-theoretic nature. We develop and explore in details how two-level type theory can be implemented in a proof assistant, providing a prototype implementation in the proof assistant Lean. We demonstrate an application of two-level type theory by developing some results on the theory of inverse diagrams using our Lean implementation.

Extending Type Theory with Syntactic Models

SIMON BOULIER
IMT Atlantique, Nantes, France

Date: November 2018; Advisor: Nicolas Tabareau
URL: https://tinyurl.com/y8w9ndas

This thesis is about the metatheory of intuitionistic type theory. We consider variants of Martin-Löf type theory or of the Calculus of Constructions, and we pay attention to two recurring questions: the coherence of these systems and the independence of axioms with respect to them. We address this kind of problems by constructing syntactic models, which are models reusing type theory to interpret type theory.

In a first part, we introduce type theory through a minimal system with several possible extensions. We also introduce Categories with Families which are models of type theory. We recall two examples: the standard model and the setoid model.

In a second part, we introduce a particular class of models: the syntactic models given by a program translation. In this paradigm, a term is translated to a term, a type to a type and a context to a context. These models are simple in that a lot of constructions are reinterpreted by themselves (universes, contexts, . . .). We give several program translations:

- the *times bool* translations, which are used to negate some extensionality principles by adding a boolean in the right place;
- a translation implementing pattern-matching on the universe;
- several variants of parametricity.

For each one, we make precise which axioms are negated or preserved by the translation.

In a third part, we present Template-Coq, a plugin for metaprogramming in Coq. This plugin provides a reification of Coq syntax and typing derivations. We demonstrate how to use it to implement directly some syntactic models.

Last, we consider type theories with two equalities: a strict one and a univalent one. We propose a re-reading of works of Coquand et. al. and of Orton and Pitts on the cubical model by introducing degenerate fibrancy. This notion of fibrancy has a defect: the universe of degenerately fibrant types is not univalent. Nonetheless, degenerate fibrancy seems worthy of attention, in particular because it admits a fibrant replacement. The fibrant replacement can be used to interpret some Higher Inductive Types and to define a model structure on the universe. This last result can be reinterpreted as the construction of a model structure on the category of cubical sets.

Unified Notions of Generalised Monads and Applicative Functors

JAN BRACKER
University of Nottingham, UK

Date: September 2018; Advisor: Henrik Nilsson
URL: https://tinyurl.com/yb6cyhtf

Monads and applicative functors are staple design patterns to handle effects in pure functional programming, especially in Haskell with its built-in syntactic support. Over the last decade, however, practical needs and theoretical research have given rise to generalisations of monads and applicative functors. Examples are graded, indexed and constrained monads. The problem with these generalisations is that no unified representation of standard and generalised monads or applicatives exists in theory or practice. As a result, in Haskell, each generalisation has its own representation and library of functions. Hence, interoperability among the different notions is hampered and code is duplicated.

To solve the above issues, I first survey the three most wide-spread generalisations of monads and applicatives: their graded, indexed and constrained variations. I then examine two approaches to give them a unified representation in Haskell: polymonads and supermonads. Both approaches are embodied in plugins for the Haskell compiler GHC that address most of the identified concerns. Finally, I examine category theory and propose unifying categorical models that encompass the three discussed generalisations together with the standard notions of monad and applicative.

Single-Assignment Program Verification

CLÁUDIO BELO LOURENÇO
Universidade do Minho, Portugal

Date: July 2018; Advisor: Jorge Sousa Pinto
URL: https://tinyurl.com/y9bn8yhm

Many *program verification* tools rely on the translation of code annotated with properties into an intermediate *single-assignment* (SA) form (in a more or less explicit way), and then on an algorithm that generates *verification conditions* (VCs) from it. In this thesis, we revisit two major methods that are widely used to produce VCs for SA programs: *predicate transformers* (used mostly by *deductive verification* tools) and the *conditional normal form transformation* (used in *bounded model checking of software*). Different aspects in which the methods differ are identified and combined to produce new hybrid VC generators; the resulting algorithms form what we call the *VCGen cube*, which we propose as a framework for synthesizing and comparing VC generators.

At the theoretical level we propose two fully proved verification frameworks based on the translation into SA and subsequent generation of VCs. On one hand we formalize program verification based on the translation of *While* programs annotated with *loop invariants* into an *iterating SA* language with a dedicated iterating construct. *Soundness* and *completeness* proofs are given for the entire workflow, including the translation of annotated programs into iterating SA form. The formalization is based on a *program logic* that we show to be *adaptation-complete*.

On the other hand we formally define an iteration-free SA language with *assume*, *assert*, and *exceptions*, and introduce a program logic for this language which allows us to prove the soundness and completeness of the VCGen cube. A verification framework based on a generic translation of programs into (iteration-free) SA form is then proposed, and the entire workflow is proved to be sound and complete. We also suggest a concrete SA translation that transforms annotated loops into assumes and asserts to check that the invariants are valid and preserved during the iterations.

Finally, we compare the VC generators empirically, both for the LLVM intermediate representation, and in the context of the Why3 deductive verification tool. Although the results do not indicate absolute superiority of any given method, they do allow us to identify interesting trends.

Subtyping in Signatures

GEORGIANA ELENA LUNGU
Royal Holloway, University of London, UK

Date: September 2018; Advisor: Zhaohui Luo
URL: https://tinyurl.com/ybtsfmms

Type theories with canonical objects like Martin Lof's Type Theory or Luo's UTT have increasingly gained popularity in the last decades due to their usage in proof assistants, formal semantics of natural language and formalization of mathematics. The main purpose of this work is to explore a new way of introducing coercive subtyping in such systems which facilitates the representation of some practical notions of subtyping.

Introducing subtyping in dependent type theories is not straightforward when the preservation of properties like canonicity and subject reduction is also desired. Previous research showed how such properties are affected by subsumptive subtyping and offered an alternative in the form of coercive subtyping introduced by enriching the system with a set of coercive subtyping judgements. Here I introduce a new way of adding coercive subtyping, specifically by annotating certain functions in assumptions, arguing that this is more handy to represent practical cases. This system is also closer to the programming model of proof assistants like Coq where coercions are annotated as such at the assumption level.

Assumptions in Type Theory are represented as either contexts, which are sequences of membership entries for variables that bear abstraction and substitution or signatures, which are sequences of memberships entries for constants for which abstraction and substitution are not available. I shall use signatures as an environment for subtyping assumptions. I will prove that this system is well behaved, in that it is only abbreviational to the original system, by considering its relation with the previous version of coercive subtyping which was already proved to be well behaved.

To demonstrate the ability of the system to argue about practical situations, I will present three case studies. The first one studies the relationship between a subsumptive subtyping system and coercive subtyping. The second case study discusses how Russell-style universe inclusions, as found in Homotopy Type Theory, can be understood as coercions in a system with Tarski-style hierarchy. And the last discussion is the need to treat injectivity as an assumption in order to capture faithfully some notions of subtyping which are based on or generalize inclusion.

JIT-based Cost Models for Adaptive Parallelism

JOHN MAGNUS MORTON
University of Glasgow, UK

Date: August 2018; Advisor: Phil Trinder and Patrick Maier
URL: https://tinyurl.com/yaksvjtz

The work in this thesis form part of the AJITPar project's *Adaptive Skeleton Library* (ASL) that provides a distributed-memory master-worker implementation of a set of Algorithmic Skeletons for *Pycket*, a tracing just-in-time compiled implementation of the Racket language. As part of ASL, this work presents a novel approach the problem of parallel *performance portability*.

The Pycket compiler is extended to enable minimal-overhead runtime access to JIT traces. A low cost, dynamic computation cost model for estimating the runtime of JIT compiled Pycket programs, Γ, is developed and validated. This is believed to be the first such model. The cost model predicts execution time based on the Pycket JIT instructions present in compiled JIT traces. Linear regression is used to determine the weightings for the abstract cost model from execution time measurements and trace data of 41 benchmarks. A linear relationship between the actual computational cost for a task, and that predicted by Γ for five benchmarks on two hardware platforms is demonstrated.

The design and iterative development of a cost model, K, that predicts the serialisation, deserialisation, and network send times of spawning a task in ASL is presented. Linear regression of communication timings are used to determine the appropriate weighting parameters for each. K is shown to be valid for predicting arbitrary data structures by demonstrating an additive property of the model. K is validated by showing a linear relationship between the combined predicted costs of the simple types in aggregated data structures, and measured communication time. This validation is performed on five benchmarks on two hardware platforms.

Finally, a low cost dynamic cost model, T, that predicts good ASL task sizes by combining information from the computation and communication cost models is developed and validated. The key insight in this model is to balance the communications cost on the master node with the computational and communications cost on the worker nodes. T is tested using six benchmarks, and it is shown to more accurately predict the optimal task size, reducing total program runtimes when compared with the default ASL prototype.

Towards Live Programming Environments for Statically Verified JavaScript

CHRISTOPHER SCHUSTER
University of California, Santa Cruz, USA

Date: December 2018; Advisor: Cormac Flanagan
URL: https://tinyurl.com/ycynhykh

This dissertation includes contributions to both live programming and program verification and explores how programming environments can be designed to leverage benefits of both concepts in an integrated way.

Programming environments assist users in both writing program code and understanding program behavior. A fast feedback loop can significantly improve this process. In particular, live programming provides continuous feedback for live code updates of running programs. This idea can also be applied to program verification. In general, verifiers statically check programs based on source code annotations such as invariants, pre- and postconditions. However, verification errors are often hard to understand, so programming environment integration is crucial for supporting the development process.

The research for this dissertation involved the implementation of esverify, a program verifier for JavaScript, as well as prototype implementations of multiple programming environments. These implementations demonstrate potential benefits and limitations of proposed solutions and enable empirical evaluation with case and user studies. Additionally, the proposed designs were formally defined in order to explain the core idea in a concise way and to prove properties independent of concrete specifics of existing systems and programming languages.

The resulting systems represent possible solutions in a vast design space with various contributions. The research on live programming showed that a programming model that separates event handling from output rendering enables not only live code updates but also runtime version control and programming-by-example. For program verification, esverify represents a novel approach for static verification of both higher-order functional programs and dynamically-typed programming idioms. esverify can verify nontrivial algorithms such as MergeSort and a formal proof in the Lean theorem prover shows that its verification rules are sound. Finally, a programming environment based on esverify supports inspection and live edits of verification conditions including step-by-step debugging of automatically generated tests that serve as executable counterexamples. As part of a user study, participants used these features effectively to solve programming tasks and generally found them to be helpful or potentially helpful.

Contributions in Programming Languages Theory: Logical Relations and Type Theory

AMIN TIMANY
KU Leuven, Belgium

Date: May 2018; Advisor: Bart Jacobs and Frank Piessens
URL: https://tinyurl.com/ychqmrjw

Software systems are ubiquitous. Failure in safety- and security-critical systems can be catastrophic. Hence, it is crucial to ensure that safety- and security-critical software systems are correct. Types play an important role in helping us achieve this goal. They help compilers check for (some) programmer's mistakes. They also form the basis of a group of proof assistants. A proof assistant is a software that allows for formalization and mechanization of mathematics, including the theory of types, theory of programming languages, program verification, etc. In this thesis we contribute to the study of programming languages and type theory.

In the first part of this thesis we formalize category theory in Coq and extend the cumulativity (subtyping relation) of Coq to inductive types. This novel extension to the type theory of Coq is integrated into the proof assistant Coq as of the official release of Coq 8.7.

In the second part of this thesis we develop logical relations models for a number of programming languages for proving type soundness and equivalence of programs for programming languages with advanced features, e.g., concurrency, impredicative polymorphism, continuations, etc. We use our logical relations models, among other things, to establish the equivalence of concurrent counter and stack modules. One of the main results of this thesis is a logical relations model which establishes proper encapsulation of state in STLang, a programming language featuring a Haskell-style ST monad which is used to encapsulate state. This problem was open for almost two decades. We solve this problem by showing that certain program equivalences hold in the presence of the ST monad that would not hold if the state was not properly encapsulated.

It is well known that developing and working with logical relations models for advanced type systems such as those we study is very intricate. We mitigate this issue by working in the Iris program logic. Working in Iris allows us to work at a higher level of abstraction and thus avoid the usual intricacies. Furthermore, we take advantage of Iris's formalization to formalize our logical relations models and their applications in Coq.