JFP **32**, e17, 9 pages, 2022.
doi:10.1017/S0956796822000144

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 eight 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

Interoperability Through Realizability: Expressing High-Level Abstractions using Low-Level Code

DANIEL BAKER PATTERSON
Northeastern University, USA

Date: August 2022; Advisor: Amal Ahmed
URL: https://tinyurl.com/2brw27kp

Large software systems are, inevitably, multi-lingual. This arises for complex socio-historical reasons, as large systems persist for years or decades, while the people working on them and the languages, libraries, and tools available to them change. Looking to these systems, I identify the *interoperability challenge*: that it is more difficult for programmers to reason about multi-lingual systems than about single-language programs. A corollary is that many of the key theorems about languages are proven in the absence of interoperability, reality notwithstanding.

In this dissertation, I identify realizability models as a key tool for addressing the interoperability challenge. Realizability models, which use target-level behavior to inhabit source types, allow the behavior of disparate source languages to be brought together. In doing so, we can recover the type of formal language-based reasoning critical to proving universal properties upon which programmers rely. In this dissertation, the property on which we focus is type soundness, which we explore through a variety of case studies and via two different interoperability mechanisms. The first mechanism, which models how typical foreign-function interfaces work, allows foreign values to be imported at existing types. Realizability models are used to demonstrate the soundness of the conversions that happen at the boundaries. The second mechanism, which better models how programmers wish interoperation worked, allows foreign code to be imported at novel types, thus allowing new behavior to be brought in. Even as the source-level mechanism is quite different between these two approaches, the underlying realizability models are similar, underscoring the central thesis: that such realizability models are an effective way of reasoning about cross-language interoperation.

On Reasonable Space and Time Cost Models for the λ-Calculus

GABRIELE VANONI
Università di Bologna, Italy

Date: June 2022; Advisor: Ugo Dal Lago
URL: https://tinyurl.com/5fev5669

The λ-calculus is considered the paradigmatic model for functional programming languages, and it is based on a rewriting mechanism. It comes with a beautiful mathematical theory, which has been studied and improved for more than 80 years. Rewriting expressions is common in computer science, but it is not the way in which programs are executed by the hardware. While the semantics of programs remains unaltered during compilation, the use of resources, in particular time and space, is more difficult to track.

Slot and van Emde Boas Invariance Thesis states some requirements for a computational model to be *reasonable*. The rationale is that under the Invariance Thesis, complexity classes such as LOGSPACE and PTIME, become robust, i.e. machine independent. The number of rewriting steps has been proved a time reasonable cost model for the λ-calculus in the majority of the interesting cases, e.g. in the call-by-name/value/need cases. Instead, there are very few results about reasonable *space* cost models.

In this dissertation, we tackle this (mostly open) problem from different perspectives. We start by considering an unusual evaluation mechanism for the λ-calculus, based on Girard's Geometry of Interaction, that was conjectured to be the key ingredient to obtain a space reasonable cost model. By a fine complexity analysis of this schema, based on new variants of non-idempotent intersection types, we disprove this conjecture. Then, we consider a variant over Krivine's abstract machine, a standard evaluation mechanism for the call-by-name λ-calculus, optimized for space complexity. We prove that the space consumed by this machine *is* a reasonable space cost model, which, for the first time, is able to account also for sub-linear space complexity. Moreover, we transfer this result to the call-by-value case. Finally, we provide an intersection type system that characterizes compositionally this new reasonable space measure in the abstract, without any reference to the machine.

Expressing Predicate Subtyping in Computational Logical Frameworks

GABRIEL HONDET
École Normale Supérieure Paris-Saclay, France

Date: September 2022; Advisor: Frédéric Blanqui and Gilles Dowek
URL: https://tinyurl.com/5n9ben9n

Safe programming as well as most proof systems rely on typing. The more a type system is expressive, the more these types can be used to encode invariants which are therefore verified mechanically through type checking procedures. Dependent types extend simple types by allowing types to depend on values. For instance, it allows to define the types of lists of a certain length. Predicate subtyping is another extension of simple type theory in which types can be defined by predicates. A predicate subtype, usually noted $\{x : A \mid P(x)\}$, is inhabited by elements t of type A for which $P(t)$ is true. This extension provides an extremely rich and intuitive type system, which is at the heart of the proof assistant PVS, at the cost of making type checking undecidable.

This work is dedicated to the encoding of predicate subtyping in Dedukti: a logical framework with computation rules. We begin with the encoding of explicit predicate subtyping for which the terms in $\{x : A \mid P(x)\}$ and terms of A are syntactically different. We show that any derivable judgement of predicate subtyping can be encoded into a derivable judgement of the logical framework. Predicate subtyping, is often used implicitly: with no syntactic difference between terms of type A and terms of type $\{x : A \mid P(x)\}$. We enrich our logical framework with a term refiner which can add these syntactic markers. This refiner can be used to refine judgements typed with implicit predicate subtyping into explicited judgements. The proof assistant PVS uses extensively predicate subtyping. We show how its standard library can be exported to Dedukti. Because PVS only store proof traces rather than complete proof terms, we sketch in the penultimate section a procedure to generate complete proof terms from these proof traces. The last section provides the architecture of a repository dedicated to the exchange of formal proofs. The goal of such a repository is to categorise and store proofs encoded in Dedukti to promote interoperability.

Bidirectional Typing for the Calculus of Inductive Constructions

MEVEN LENNON-BERTRAND
Université de Nantes, France

Date: June 2022; Advisor: Nicolas Tabareau
URL: https://tinyurl.com/4axpzf5u

Over their more than 50 years of existence, proof assistants have established themselves as tools guaranteeing high trust levels in many applications. Yet, due to their ever increasing complexity, the historical solution of relying on a small, trusted kernel is not enough any more to avoid critical bugs while moving forward. But proof assistants have been used for decades to certify program correctness, why not *their own*? This is the ambition of the MetaCoq project, which aims at providing the first kernel for a real-life proof assistant, Coq, that is formally proven correct, in Coq itself. Don't trust the kernel any more, only its correctness proof! To that aim, this thesis studies the bidirectional structure which underpins the typing algorithm implemented by the kernel of Coq, in the context of the Calculus of Inductive Constructions on which said kernel is founded.

It first considers this bidirectional structure from a theoretical point of view. It exposes a bidirectional presentation of CIC, together the general discipline that led to it. Follow a roof of equivalence between this presentation and the standard one. This equivalence is then used to establish properties of CIC that are hard to obtain in the standard setting– existence of principal types, and strengthening–, showing the power of this approach in studying the meta-theory of (dependent) type systems.

The second part sets on to formalize the idea of the first one in the setting of the MetaCoq project, and to use them to show correctness of the kernel. The formalized bidirectional structure supplies an intermediate between the high-level specification and the algorithm, which is key in order to prove that the kernel is complete.

Finally, the last part considers the question of designing an extension of CIC along ideas from gradual typing, with the aim of incorporating some form of dynamic typechecking to bring more flexibility to development in Coq. The bidirectional structure is once again crucial, as the characteristics of gradual typing — in particular the way it relaxes conversion — make it impossible to base this extension on the standard presentations of CIC.

Mechanized Reasoning about "how" using Functional Programs and Embeddings

YAO LI
University of Pennsylvania, USA

Date: August 2022; Advisor: Stephanie Weirich
URL: https://tinyurl.com/m7uuxwas

Embedding describes the process of encoding a program's syntax and/or semantics in another language—typically a theorem prover in the context of mechanized reasoning. Among different embedding styles, deep embeddings are generally preferred as they enable the most faithful modeling of the original language. However, deep embeddings are also the most complex, and working with them requires additional effort. In light of that, this dissertation aims to draw more attention to alternative styles, namely shallow and mixed embeddings, by studying their use in mechanized reasoning about programs' properties that are related to "how". More specifically, I present a simple shallow embedding for reasoning about computation costs of lazy programs, and a class of mixed embeddings that are useful for reasoning about properties of general computation patterns in effectful programs. I show the usefulness of these embedding styles with examples based on real-world applications.

Practical Heterogeneous Unification for Dependent Type Checking

VÍCTOR LÓPEZ JUAN
Chalmers University of Technology, Sweden

Date: December 2021; Advisor: Nils Anders Danielsson and Andreas Abel
URL: https://tinyurl.com/2p8b93ky

Dependent types can specify in detail which inputs to a program are allowed, and how the properties of its output depend on the inputs. A program called the type checker assesses whether a program has a given type, thus detecting situations where the implementation of a program potentially differs from its intended behaviour. When using dependent types, the inputs to a program often occur in the types of other inputs or in the type of the output. The user may omit some of these redundant inputs when calling the program, expecting the type-checker to infer those subterms automatically.

Some type-checkers restrict the inference of missing subterms to those cases where there is a provably unique solution. This makes the process more predictable, but also limits the situations in which the omitted terms can be inferred; specially when considering that whether a unique solution exists is in general an undecidable problem. This restriction can be made less limiting by giving flexibility to the type-checker regarding the order in which the missing subterms are inferred. The type-checker can then use the information gained by filling in any one subterm in order to infer others, until the whole program has been type-checked. However, this flexibility may in some cases lead to ill-typed subterms being inferred, breaking internal invariants of the type-checker and causing it to crash or loop. The type checker could mitigate this by consistently rechecking the type of each inferred subterm, but this might incur a performance penalty.

An approach by Gundry and McBride (2012) called twin types has the potential to afford the desired flexibility while preserving well-typedness invariants. However, this method had not yet been tested in a practical setting. In this thesis we streamline the method of twin types in order to ease its practical implementation, justify the correctness of our modifications, and then implement the result in an established dependently-typed language called Agda. We show that our implementation resolves certain existing bugs in Agda while still allowing a wide range of examples to be type-checked, and achieves this without heavily impacting performance.

Formally Verified Bundling and Appraisal of Layered Attestation Protocols

ADAM PETZ
University of Kansas, USA

Date: June 2022; Advisor: Perry Alexander
URL: https://tinyurl.com/4s9pytsk

Remote attestation is a technology for establishing trust in a remote computing system. Core to the integrity of the attestation mechanisms themselves are components that orchestrate, cryptographically bundle, and appraise measurements of the target system. Copland is a domain-specific language for specifying attestation protocols that operate in diverse, layered measurement topologies. In this work we formally define and verify the Copland Virtual Machine alongside a dual generalized appraisal procedure. Together these components provide a principled pipeline to execute and bundle arbitrary Copland-based attestations, then un-bundle and evaluate the resulting evidence for measurement content and cryptographic integrity. All artifacts are implemented as monadic, functional programs in the Coq proof assistant and verified with respect to a Copland reference semantics that characterizes attestation-relevant event traces and cryptographic evidence structure. Appraisal soundness is positioned within a novel end-to-end workflow that leverages formal properties of the attestation components to discharge assumptions about honest Copland participants. These assumptions inform an existing model-finder tool that analyzes a Copland scenario in the context of an active adversary attempting to subvert attestation. An initial case study exercises this workflow through the iterative design and analysis of a Copland protocol and accompanying security architecture for an Unmanned Air Vehicle demonstration platform. We conclude by instantiating a more diverse benchmark of attestation patterns called the *Flexible Mechanisms for Remote Attestation*, leveraging Coq's built-in code synthesis to integrate the formal artifacts within an executable, Haskell-based attestation environment.

TopHat: Task-Oriented Programming with Style

TIM STEENVOORDEN
Radboud University, The Netherlands

Date: October 2022; Advisor: Rinus Plasmeijer and Johan Jeuring
URL: `https://tinyurl.com/ymyphhaa`

Everywhere where people collaborate, workflows are omnipresent. Almost every corporation, government, or institution needs a way to coordinate people and machines, following prescribed ways of working. We call this units of work *tasks*. There is added benefit of specifying tasks, as one can create computer programs to aid us during task execution. But to do this, there should be sufficient formality in the workflow specification to be able to create computer applications out of it.

Task-oriented programming (TOP) tries to close the gap between faithfully and understandably modelling real world tasks and the creation of applications that support such workflows. It does so, while taking away the recurring programming activities when creating distributed and fault tolerant applications with persistent data and interactive user interfaces.

In this thesis we introduce TopHat, a formal language faithful to the core principles of TOP. TopHat can be used as a specification language to describe workflows. At the same time, TopHat is fully mathematically formalised. This means that we can prove rigorous statements and properties about the language itself, as well as the workflows written in it.

We start by capturing the intuition of TopHat and describe its main components. We define its language syntax and give semantic meaning to TopHat programs. Main design goal is to keep a clear distinction between basic programming expressions, tasks, and the interaction with end users. This results in three clearly separated layers of semantics, together with static observations on tasks.

Then, we present three practical applications of the TopHat language. We show how TopHat workflows can be visualised, and introduce a structured way to gradually develop such visualisations. By employing symbolic execution, we can automatically verify assertions about TopHat programs. We utilise this symbolic engine to generate next step hints for end users.

Finally, we prove basic language safety properties for TopHat, and consistency between our concrete and our symbolic semantics. Also, we answer the question when two tasks are equivalent.

The TopHat language, together with its applications and theoretical properties, forms a coherent foundation to which other formal methods can be applied to reason about task-oriented programs.