JFP **32**, e2, 6 pages, 2022.
doi:10.1017/S0956796821000290

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 five 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

Algebraic Information Effects

CHAO-HONG CHEN
Indiana University, USA

Date: August 2021; Advisor: Amr Sabry
URL: https://tinyurl.com/3a9tsrpy

From the informational perspective, programs that are usually considered as pure have effects, for example, the simply typed lambda calculus is considered as a pure language. However, β–reduction does not preserve information and embodies information effects. To capture the idea about pure programs in the informational sense, a new model of computation — reversible computation was proposed. This work focuses on type-theoretic approaches for reversible effect handling. The main idea of this work is inspired by compact closed categories. Compact closed categories are categories equipped with a dual object for every object. They are well-established as models of linear logic, concurrency, and quantum computing. This work gives computational interpretations of compact closed categories for conventional product and sum types, where a negative type represents a computational effect that "reverses execution flow" and a fractional type represents a computational effect that "allocates/deallocates space".

Domain-Specific Languages for Ad Hoc Data Processing

JONATHAN DILORENZO
Cornell University, USA

Date: December 2020; Advisor: John Nathan Foster
URL: https://tinyurl.com/2p88kunn

Ad hoc data is everywhere. There are many data formats in use, which, due to the specificity of their domain, lack the processing tools that we otherwise take for granted. We call such data formats, and the data that they represent, *ad hoc*. Ad hoc data is usually stored in file systems and can be organized into larger structures that we call filestores, collections of files and folders along with the properties between them. This dissertation supports the usage of ad hoc filestores. We design domain-specific languages for processing filestores with increasingly complex requirements, building on previous work on PADS and Forest. These existing systems accept declarative specifications of ad hoc data formats, as single files and filestores respectively. From these specifications, they generate tools for loading, storing, and validating the data. Unfortunately, Forest does not adequately deal with large filestores, or cost control in general. Nor does it offer support for correctly managing concurrent operations, which are common in file systems. This dissertation offers solutions to these problems. We first introduce Incremental Forest, a domain-specific language and system that enables incremental processing of filestores. This language offers a new mechanism, a delay construct, for explicitly controlling the costs of loading and storing in filestores. Incremental Forest comes with a customizable cost model, which guarantees that a wide class of costs monotonically decrease as delays increase. Our next system, Transactional Forest, removes the delays from Incremental Forest, opting to use a new interface language and abstraction, which offer automatic incrementality. Additionally, Transactional Forest leverages this abstraction, a zipper, to provide simple, provably correct serializable transactions using optimistic concurrency control. Finally, the Zipper File System goes beyond designing a domain-specific language, targeting the file system itself to provide deeper control with respect to other users. The Zipper File System uses the ideas from Transactional Forest to provide serializable transactions in a zipper-based file system. There is a translation from POSIX that theoretically allows standard applications to be run without modification. Taken together, these systems use domain-specific languages to enable users to efficiently and correctly manage ad hoc filestores in concurrent settings.

Efficient Parsing with Derivatives and Zippers

ROMAIN EDELMANN
École Polytechnique Fédérale de Lausanne, Switzerland

Date: July 2021; Advisor: Viktor Kunčak
URL: https://tinyurl.com/y5dtdb6f

In this thesis, I present a declarative framework for parsing that explicitly integrates data elaboration. Within the framework of the thesis, I present parsing algorithms that are based on the concept of Brzozowski's derivatives. Derivative-based parsing algorithms present several advantages: they are elegant, amenable to formal reasoning, and easy to implement. Unfortunately, the performance of these algorithms in practice is often not competitive with other approaches. In this thesis, I show a general technique inspired by Huet's Zipper to greatly enhance the performance of derivative-based algorithms, and I do so without compromising their elegance, amenability to formal reasoning, or ease of implementation. First, I present a technique for building efficient tokenisers that is based on Brzozowski's derivatives and Huet's zipper and that does not require the usual burdensome explicit conversion to automata. I prove the technique is correct in Coq and present SILEX, a Scala lexing library based on the technique. I demonstrate that the approach is competitive with state-of-the-art solutions. Then, I present a characterisation of LL(1) languages based on the concept of should-not-follow sets. I present an algorithm for parsing LL(1) languages with derivatives and zippers. I show a formal proof of the algorithm's correctness and prove its worst-case linear-time complexity. I show how the LL(1) parsing with derivatives and zippers algorithm corresponds to the traditional LL(1) parsing algorithm. I then present SCALL1ON, a Scala parsing combinators library for LL(1) languages that incorporates the LL(1) parsing with derivatives and zippers algorithm. I present an expressive and familiar combinator-based interface for describing LL(1) languages. I present techniques that help precisely locate LL(1) conflicts in user code. I discuss several advantages of the parsing with derivatives approach within the context of a parsing library. I also present SCALL1ON's enumeration and pretty-printing features and discuss their implementation. Through a series of benchmarks, I demonstrate the good performance and practicality of the approach. Finally, I present how to adapt the LL(1) parsing with derivatives and zippers algorithm to support arbitrary context-free languages. I show how the adapted algorithm corresponds to general parsing algorithms, such as Earley's parsing algorithm.

Understanding the Interaction Between Elaboration And Quotation

MATTHEW PICKERING
University of Bristol, UK

Advisor: Nicolas Wu
Date: March 2021; URL: https://tinyurl.com/2p8saaed

Multi-stage programming languages have long promised programmers the means to program libraries with specific performance guarantees. Despite this, the adoption into mainstream high-level languages such as Haskell, Scala and OCaml has been very slow. In particular, our focus, Typed Template Haskell has been implemented for a number of years but the ecosystem has failed to adopt the multi-stage ideas. The situation hasn't been helped by a number of soundness and expressivity issues with the current implementation.

In this thesis we tackle the problem of soundly combining multi-stage features with other features as commonly found in modern programming languages. The main contribution is the design and implementation of a language which combines multiple stages with implicit arguments and an elaboration phase. The goal is to provide a firmer foundation for future implementations and to enable library writers to use multi-stage features with confidence in conjunction with the rest of the Haskell language.

Proving Confidentiality and Its Preservation Under Compilation for Mixed-Sensitivity Concurrent Programs

ROBERT ABELLA SISON
University of New South Wales, Australia

Date: November 2020; Advisor: Carroll Morgan, Toby Murray and Kai Engelhardt
URL: https://tinyurl.com/2ucyjp4k

Here, I pose the thesis that *proving noninterference and its preservation by a compiler is feasible for mixed-sensitivity concurrent programs*. Software does not always have the luxury of limiting itself to single-threaded computation with resources statically dedicated to each user to ensure the confidentiality of their data. Prior work therefore presented formal methods for proving and preserving the strictest kind of confidentiality property, *noninterference*, for *mixed-sensitivity concurrent programs*: a term I coin to describe those programs that might reuse memory shared between their threads to hold data of different sensitivity levels at different times. Although these methods addressed challenges in formalising the *value-dependent* coordination of such *mixed-sensitivity reuse* under the impact of *concurrency*, their practicality remained unclear: Could they be used to prove noninterference for any nontrivial mixed-sensitivity concurrent program in its entirety? Furthermore, could any compiler be verified to preserve the needed guarantees to the compiled code?

To support this claim, I prove for the first time both (1) noninterference for a nontrivial mixed-sensitivity concurrent program, modelling a real-world use case, and (2) its preservation by a compiler down to an assembly-level model. This main result rests on two major contributions. First, I demonstrate how programming-language designers can make reasoning on each thread sufficient to prove noninterference for such programs, by supplying synchronisation primitives (here, mutex locks for a generic imperative language) and proving they maintain as invariant the necessary requirements. Second, I demonstrate how compiler developers can make *confidentiality-preserving refinement* a feasible target for verification, by using a decomposition principle to prove that a compiler (here, from that imperative language to a generic RISC-style assembly language) establishes it for mixed-sensitivity concurrent programs. Thus, per-thread reasoning proves noninterference for the case study, and the verified compiler preserves it to assembly automatically. All my results are formalised and proved in the Isabelle/HOL interactive proof assistant.

My work paves the way for more fully featured programming languages and their compilers, in replicating these results, to raise the typical level of assurance readily offered by developers of multithreaded software responsible for data of multiple sensitivity levels.