JFP **24** (4): 474–522, 2014.
doi:10.1017/S0956796814000215 First published online 27 August 2014

PhD Abstracts

GRAHAM HUTTON
University of Nottingham, UK
(*e-mail:* graham.hutton@nottingham.ac.uk)

Many students complete PhDs in functional programming each year, but there is currently no common location in which to promote and advertise the resulting work. The Journal of Functional Programming would like to change that!

As a service to the community, JFP is launching a new feature, in the form of a regular publication of abstracts from PhD dissertations that were completed during the previous year. To start this new feature off, we have reached back three years for the first round of abstracts. The abstracts are freely available on the JFP website, i.e. not behind any paywall, and do not require any transfer for copyright, merely a license from the author. A dissertation is eligible 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 37 abstracts in this first round, and hope that JFP readers will find many interesting dissertations in this collection that they might not otherwise have seen. If a student or advisor would like to submit a dissertation abstract for publication in this series, please contact the editor for further details.

Graham Hutton
PhD Abstract Editor

Flow-sensitivecontrol-flow analysis in linear-log time

MICHAEL D. ADAMS
Indiana University, USA

Date: October 2011; Advisor: R. Kent Dybvig
URL: http://search.proquest.com/docview/914424665

The flexibility of dynamically typed languages such as JavaScript, Python, Ruby, and Scheme comes at the cost of run-time type checks. Some of these checks can be eliminated via control-flow analysis. However, traditional control-flow analysis (CFA) is not ideal for this task as it ignores flow-sensitive information that can be gained from dynamic type predicates, such as JavaScript's `instanceof` and Scheme's `pair?`, and from type-restricted operators, such as Scheme's `car`. Yet, adding flow-sensitivity to a traditional CFA worsens the already significant compile-time cost of traditional CFA. This makes it unsuitable for use in just-in-time compilers.

In response, this dissertation presents a fast, flow-sensitive type-recovery algorithm based on the linear-time, flow-insensitive sub-0CFA. The algorithm has been implemented as an experimental optimization into Chez Scheme compiler, where it has proven to be effective, justifying the elimination of about 60% of run-time type checks in a large set of bench-marks. The algorithm processes on average over 100,000 lines of code per second and scales well asymptotically, running in only $O(n \log n)$ time. This compile-time performance and scalability is achieved through a novel combination of data structures and algorithms.

Finding the lazy programmer's bugs

TRISTAN ALLWOOD
Imperial College London, UK

Date: January 2011; Advisor: Susan Eisenbach
URL: http://pubs.doc.ic.ac.uk/tristan-allwood-thesis/

Traditionally developers and testers created huge numbers of explicit tests, enumerating interesting cases, perhaps biased by what they believe to be the current boundary conditions of the function being tested. Or at least, they were supposed to.

A major step forward was the development of property testing. Property testing requires the user to write a few functional properties that are used to generate tests, and requires an external library or tool to create test data for the tests. As such many thousands of tests can be created for a single property. For the purely functional programming language Haskell there are several such libraries; for example QuickCheck, SmallCheck and Lazy SmallCheck.

Unfortunately, property testing still requires the user to write explicit tests. Fortunately, we note there are already many implicit tests present in programs. Developers may throw assertion errors, or the compiler may silently insert runtime exceptions for incomplete pattern matches.

We attempt to automate the testing process using these implicit tests. Our contributions are in four main areas: (1) We have developed algorithms to automatically infer appropriate constructors and functions needed to generate test data without requiring additional programmer work or annotations. (2) To combine the constructors and functions into test expressions we take advantage of Haskell's lazy evaluation semantics by applying the techniques of needed narrowing and lazy instantiation to guide generation. (3) We keep the type of test data at its most general, in order to prevent committing too early to monomorphic types that cause needless wasted tests. (4) We have developed novel ways of creating Haskell case expressions to inspect elements inside returned data structures, in order to discover exceptions that may be hidden by laziness, and to make our test data generation algorithm more expressive.

In order to validate our claims, we have implemented these techniques in Irulan, a fully automatic tool for generating systematic black-box unit tests for Haskell library code. We have designed Irulan to generate high coverage test suites and detect common programming errors in the process.

The mechanics of the grammatical framework

KRASIMIR ANGELOV
Chalmers University of Technology, Sweden

Date: February 2012; Advisor: Aarne Ranta
URL: http://www.cse.chalmers.se/ krasimir/phd-thesis.pdf

Grammatical Framework (GF) is a well known theoretical framework and a mature programming language for description of natural languages. The GF community is now growing rapidly and the range of applications is expanding. Within the framework, there are computational resources for 26 languages created from different people in different organizations. The coverage of the different resources varies but there are complete morphologies and grammars for at least 18 languages. This advancement would not be possible without the continuous development of the GF compiler and interpreter.

The demand for efficient and portable execution model for GF lead to major changes in both the compiler and the interpreter. We developed a new low-level representation called Portable Grammar Format (PGF) which is simple enough for an efficient interpretation. Since it was already known that a major fragment of GF is equivalent to Parallel Multiple Context-Free Grammar (PMCFG), we designed PGF as an extension which adds to PMCFG distinctive features of GF such as multilingualism, higher-order abstract syntax, dependent types, etc. In the process we developed novel algorithms for parsing and linearization with PMCFG and a framework for logical reasoning in first-order type theory where the proof search can be constrained by the parse chart.

This monograph is the detailed description of the engine for efficient interpretation of PGF and is intended as a reference for building alternative implementations or as a foundation for the future development of PGF.

Modular implementation of programming languages and a partial-order approach to infinitary rewriting

PATRICK BAHR
University of Copenhagen, Denmark

Date: December 2012; Advisor: Fritz Henglein
URL: http://tinyurl.com/kca9pzk

In this dissertation we study two independent areas of research: implementation of programming languages on the one hand and infinitary rewriting on the other hand.

In the first part, titled *Modular Implementation of Programming Languages*, we develop techniques for implementing programming languages in a modular fashion. Within this problem domain, we focus on operations on typed abstract syntax trees with the goal of developing a framework that facilitates the definition, manipulation and composition of such operations. The result of our work is a comprehensive combinator library that provides these facilities.

What sets our approach apart is the use of recursion schemes derived from *tree automata* in order to implement operations on abstract syntax trees. In the first two chapters we illustrate the power of this approach by showcasing *tree homomorphisms* as basic building blocks for simple tree transformations. Tree homomorphisms are a very limited form of tree automata that transform the tree structure depending only on local information. Their simplicity allows us to combine them with monadic effects, manipulate and combine them in a flexible manner, and perform optimisations in the form of *deforestation*. In the second chapter, we focus on the important issue of representing variable names and variable binders using a carefully restricted form of *higher-order abstract syntax*.

In the third chapter, we move to more powerful tree automata, namely *tree transducers*. In essence, tree transducers combine tree homomorphisms with the capability of maintaining state information that is propagated downwards or upwards trough the tree structure. Usually, these more powerful automata are cumbersome to define as they combine different computational aspects: transforming the tree structure and maintaining state information. We show, however, that these automata can be constructed from simpler ones, namely tree homomorphisms and simple state machines.

In the fourth chapter, we present a comprehensive and realistic application of our library: a prototype implementation of a novel *enterprise resource planning* system built around a family of domain-specific languages that make it possible to customise the system in a highly flexible manner. The system combines several highly integrated domain-specific languages, which are implemented using our library. Due to the common underlying domain, there is a non-trivial amount of overlap between the languages. With our library we are able to reuse implementations of functionality that is shared between the different languages.

The second part of this dissertation, titled *A Partial-Order Approach to Infinitary Rewriting*, is concerned with *infinitary rewriting*, a field that studies transfinite rewrite sequences.

In infinitary rewriting one gives meaning to infinite rewrite sequences by providing a notion of convergence. That is, one defines what it means for an infinite rewrite sequence to be well-behaved and what the limit of such a well-behaved rewrite sequence is. To this end, one typically endows the term language with a metric and derives convergence from the resulting metric space.

In this dissertation, we extend the established theory of infinitary rewriting in two ways:

1. a novel approach to convergence in infinitary rewriting that replaces convergence in a metric space with the *limit inferior* in a partially ordered set;
2. extending infinitary term rewriting to infinitary term *graph* rewriting.

To facilitate our study of convergence, we formulate a common framework that abstracts from the notion of convergence, e.g. metric convergence, and the underlying objects, e.g. terms or graphs. We show that, in this abstract framework, many basic relations between termination and confluence properties known from finite reductions still hold in the infinitary setting.

We then introduce a novel notion of convergence for infinitary rewriting based on a partial order on terms. The infinitary rewriting theory is obtained by instantiating the abstract framework with the limit inferior of the partial order as the notion of convergence. We show correspondences between the established calculi based on metric convergence and the newly developed calculi based on partial orders. In particular, we show that the partial order approach is a conservative extension of the metric approach. Moreover, we find that the partial order-based calculi have better confluence and normalisation properties than the metric-based ones. In light of these results we argue that the partial order approach is superior to the established metric approach.

To compare the two approaches further, we consider so-called Böhm extensions, which extend term rewrite systems with rules that contract certain terms to $\bot$. Such extensions were originally devised for infinitary term rewriting in order to re-obtain normalisation and convergence properties, which are lost due to infinite rewrite sequences. We show that metric infinitary term rewriting with Böhm extensions coincides with partial order infinitary term rewriting.

Finally, we give the first rigorous treatment of infinitary term graph rewriting. To this end we instantiate our abstract framework with term graphs. However, devising suitable notions of convergence on term graphs turns out to be non-trivial and we therefore explore different approaches. Among these different attempts we distinguish two calculi – a metric-based one and a partial order-based one. We show the appropriateness of these calculi by proving soundness and completeness properties with respect to the corresponding infinitary term rewriting calculi. Also in the setting of term graph rewriting the partial order approach shows advantages over the metric approach: the completeness property for the metric calculus is weaker than for the partial order calculus. This weakness is inherent in the metric notion of convergence and is independent of the particular choice of the metric.

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

Parallel functional programming with mutable state

LARS BERGSTROM
University of Chicago, USA

Date: June 2013; Advisor: John Reppy
URL: http://manticore.cs.uchicago.edu/papers/bergstrom-phd.pdf

Immutability greatly simplifies the implementation of parallel languages. In the absence of mutable state the language implementation is free to perform parallel operations with fewer locks and fewer restrictions on scheduling and data replication. In the Manticore project, we have achieved nearly perfect speedups across both Intel and AMD manycore machines on a variety of benchmarks using this approach.

There are parallel stateful algorithms, however, that exhibit significantly better performance than the corresponding parallel algorithm without mutable state. For example, in many search problems, the same problem configuration can be reached through multiple execution paths. Parallel stateful algorithms share the results of evaluating the same configuration across threads, but parallel mutation-free algorithms are required to either duplicate work or thread their state through a sequential store. Additionally, in algorithms where each parallel task mutates an independent portion of the data, non-conflicting mutations can be performed in parallel. The parallel state-free algorithm will have to merge each of those changes individually, which is a sequential operation at each step.

In this dissertation, we extend Manticore with two techniques that address these problems while preserving its current scalability. *Memoization*, also known as function caching, is a technique that stores previously returned values from functions, making them available to parallel threads of executions that call that same function with those same values. We have taken this deterministic technique and combined it with a high-performance implementation of a dynamically sized, parallel hash table to provide scalable performance. We have also added *mutable state* along with two execution models — one of which is deterministic — that allow the user to share arbitrary results across parallel threads under several execution models, all of which preserve the ability to reason locally about the behavior of code.

For both of these techniques, we present a detailed description of their implementations, examine a set of relevant benchmarks, and specify their semantics.

Call-by-need supercompilation

MAXIMILIAN C. BOLINGBROKE
University of Cambridge, UK

Date: July 2013; Advisor: Simon Peyton Jones and Alan Mycroft
URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-835.html

This thesis shows how supercompilation, a powerful technique for transformation and analysis of functional programs, can be effectively applied to a call-by-need language. Our setting will be core calculi suitable for use as intermediate languages when compiling higher-order, lazy functional programming languages such as Haskell.

We describe a new formulation of supercompilation which is more closely connected to operational semantics than the standard presentation. As a result of this connection, we are able to exploit a standard Sestoft-style operational semantics to build a supercompiler which, for the first time, is able to supercompile a call-by-need language with unrestricted recursive let bindings.

We give complete descriptions of all of the (surprisingly tricky) components of the resulting supercompiler, showing in detail how standard formulations of supercompilation have to be adapted for the call-by-need setting.

We show how the standard technique of generalisation can be extended to the call-by-need setting. We also describe a novel generalisation scheme which is simpler to implement than standard generalisation techniques, and describe a completely new form of generalisation which can be used when supercompiling a typed language to ameliorate the phenomenon of supercompilers overspecialising functions on their type arguments.

We also demonstrate a number of non-generalisation-based techniques that can be used to improve the quality of the code generated by the supercompiler. Firstly, we show how let-speculation can be used to ameliorate the effects of the work-duplication checks that are inherent to call-by-need supercompilation. Secondly, we demonstrate how the standard idea of rollback in supercompilation can be adapted to our presentation of the supercompilation algorithm.

We have implemented our supercompiler as an optimisation pass in the Glasgow Haskell Compiler. We perform a comprehensive evaluation of our implementation on a suite of standard call-by-need benchmarks. We improve the runtime of the benchmarks in our suite by a geometric mean of 42%, and reduce the amount of memory which the benchmarks allocate by a geometric mean of 34%.

Stream fusion: Practical shortcut fusion for coinductive sequence types

DUNCAN COUTTS
University of Oxford, UK

Date: 2011; Advisor: Oege de Moor
URL: http://tinyurl.com/k8pnqtf

In functional programming it is common practice to build modular programs by composing functions where the intermediate values are data structures such as lists or arrays. A desirable optimisation for programs written in this style is to fuse the composed functions and thereby eliminate the intermediate data structures and their associated runtime costs.

Stream fusion is one such fusion optimisation that can eliminate intermediate data structures, including lists, arrays and other abstract data types that can be viewed as coinductive sequences. The fusion transformation can be applied fully automatically by a general purpose optimising compiler. The stream fusion technique itself has been presented previously and many practical implementations exist. The primary contributions of this thesis address the issues of correctness and optimisation: whether the transformation is correct and whether the transformation is an optimisation.

Proofs of shortcut fusion laws have typically relied on parametricity by making use of free theorems. Unfortunately, most functional programming languages have semantics for which classical free theorems do not hold unconditionally; additional side conditions are required. In this thesis we take an approach based not on parametricity but on data abstraction. Using this approach we prove the correctness of stream fusion for lists – encompassing the fusion system as a whole, not merely the central fusion law. We generalise this proof to give a framework for proving the correctness of stream fusion for any abstract data type that can be viewed as a coinductive sequence and give as an instance of the framework, a simple model of arrays. The framework requires that each fusible function satisfies a simple data abstraction property. We give proofs of this property for several standard list functions.

Previous empirical work has demonstrated that stream fusion can be an optimisation in many cases. In this thesis we take a more universal view and consider the issue of optimisation independently of any particular implementation or compiler. We make a semi-formal argument that, subject to certain syntactic conditions on fusible functions, stream fusion on lists is strictly an improvement, as measured by the number of allocations of data constructors. This detailed analysis of how stream fusion works may be of use in writing fusible functions or in developing new implementations of stream fusion.

Foundations for behavioral higher-order contracts

CHRISTOS DIMOULAS
Northeastern University, USA

Date: January 2013; Advisor: Matthias Felleisen
URL: http://hdl.handle.net/2047/d20002848

Contracts are a popular mechanism for enhancing the interface of components. In the world of first-order functions, programmers embrace contracts because they write them in a familiar language and easily understand them as a pair of a pre-condition and a post-condition. In a higher-order world, contracts offer the same expressiveness to programmers but their meaning subtly differs from the familiar first-order notion. For instance, it is unclear what the behavior of dependent contracts for higher-order functions or of contracts for mutable data should be. As a consequence, it is difficult to design monitoring systems for such higher-order worlds.

In response to this problem, this dissertation investigates complete monitors, a formal framework for deciding if a contract system is correct. The intuition behind the framework is that a correct contract system should:

- mediate the exchange of values between contracted components
- and blame correctly in case of contract violations.

The framework reveals flaws in the semantics for dependent contracts from the literature and suggests a natural fix. In addition, this dissertation demonstrates the usefulness of the framework for language design with a language with contracts for mutable data and a language that mixes typed and untyped imperative programs. The final contribution is the provably correct design of a novel form of contracts, dubbed options contracts, that mix contract checking with random and probabilistic checking.

Modular proof development in ACL2

CARL EASTLUND
Northeastern University, USA

Date: January 2013; Advisor: Matthias Felleisen
URL: http://www.ccs.neu.edu/racket/pubs/dissertation-eastlund.pdf

The ACL2 theorem prover combines a first-order dialect of LISP with an automated proof engine for first-order logic. While ACL2 is logically quite powerful, it can be difficult to build and maintain large models due to its ad hoc systems for modularity, namespace management, logical encapsulation, and macro expansion. I propose a new language, Refined ACL2, extending ACL2 with expressive component and macro systems designed to accommodate large-scale proof development and flexible logical abstractions. The component system of Refined ACL2 adapts many features of ML's functors and signatures to ACL2. Components support nesting, parameterization, translucent specification, and refinement of abstract specifications with concrete definitions. Refined ACL2 inherits Racket's macro system; furthermore, macro definitions can be incorporated into component specifications.

Frontiers of multilingual grammar development

RAMONA ENACHE
University of Gothenburg, Sweden

Date: October 2013; Advisor: Aarne Ranta and Koen Claessen
URL: http://tinyurl.com/ks6ke7j

The thesis explores a number of ways for developing multilingual grammars written in GF (Grammatical Framework). GF is a type-theoretical grammar formalism, particularly suited for a multilingual setting and a dependently-typed functional programming language, usually used for natural language applications. The GF approach to language representation is inspired from compiler theory and separates a grammar into the abstract syntax (semantic interlingua for the domain), and a number of concrete syntaxes (usually corresponding to natural languages). This representation allows direct translation between any pair of concrete languages, via the abstract syntax, and a semantically-coherent representation of the domain across languages.

The goal of this work is to enhance both the coverage of the grammars, in terms of content and number of languages and to reduce the development effort by automating a larger part of the process.

The first direction in grammar development targets the creation of general language resources. These are the starting point for building domain-specific grammars for the given language. Developing resource grammars gives a good overview on the effort required and provides a solid base for subsequent experiments in automation. Our work resulted in building computational grammars for Romanian and Swedish.

A further development step is multilingual domain-specific grammar creation. The technique we employed is converting structured models into grammars, which preserves the original structure of the model as a backbone of the grammar and uses the general GF resources for a smooth multilingual verbalization of the model. The use cases considered are an upper-domain ontology, a business model and an ontology describing cultural heritage artifacts, each posing a different challenge and illustrating a different aspect of GF grammars-ontology interoperability, and its advantages on both sides.

An orthogonal approach to multilingual grammar development aims at increasing the number of languages from a domain grammar. Our solution is an example-based prototype which partially replaces grammar programming with feedback from native informants and/or SMT tools (such as Google Translate).

Last but not least, as an attempt to not only enhance GF grammars, but also use them in a novel way, we present the grammar-based hybrid system architecture combining GF grammars and SMT systems. This marks some of the first steps in using grammars for translating free text. As a side-effect of the work, we propose a technique for building bilingual GF lexicon resources from SMT phrase tables.

Ask-Elle: A Haskell tutor

ALEX GERDES
Open Universiteit Nederland, The Netherlands

Date: November 2012; Advisor: Johan Jeuring
URL: http://tinyurl.com/mceu33u

Learning to program is challenging. A first course in programming is often a major stumbling block and the results of such a course are often disappointing. There is no final answer yet to the question how programming is learned best, and what makes programming hard. There are many aspects to learning a new programming language or paradigm. One important aspect is to practise programming, by making programming exercises. A student learns from the feedback he receives on his solutions. Preferably this feedback is given while the student is doing the exercise, instead of giving feedback later. This feedback is usually given by a teacher or assistant. Giving immediate feedback becomes hard when a teacher has to help many students simultaneously. To support a teacher with this task many intelligent programming tutors have been developed.

There exist programming tutors for a number of programming languages, such as Java, Lisp, Prolog, Haskell, and many more. Evaluation studies have indicated that working with a programming tutor supporting the construction of programs is more effective when learning how to program than doing the same exercise 'on your own' using only a compiler. Using intelligent tutors requires less help from a teacher while showing the same performance on tests. The immediate feedback given by many of the tutors is to be preferred over the delayed feedback common in a classroom setting. Despite the advantages of using intelligent programming tutors, they are not widely used. Some tutors are well-developed and extensively tested in classrooms, but most haven't outgrown the research prototype phase, and are not maintained any more. Furthermore, deploying an intelligent tutor in a course is often hard for a teacher. Most teachers want to adapt or extend an intelligent programming tutor to their needs. Adding an exercise to a tutor requires investigating which strategies can be used to solve the exercise, what the possible solutions are, and how the tutor should react to behaviour that doesn't follow the desired path. All this knowledge then has to be translated into the internals of the tutor, which implies a substantial amount of work.

In this thesis we investigate the following research questions: how can we design and implement a functional programming tutor that automatically gives semantically rich feedback to students incrementally solving an exercise, to which teachers can easily add exercises, in which teachers can easily fine-tune feedback?

We have built a functional programming tutor for Haskell, named Ask-Elle, in which we address these research questions. Ask-Elle is an interactive system that supports the stepwise development of simple functional programs and targets

beginning computer science students. A notable feature of our tutor is that the hints and feedback given at intermediate steps are derived automatically from teacher-specified annotated solutions and non-solutions for a problem. This reduces the work required for using the tutor, and allows a teacher to use his favourite exercises. Furthermore, the order in which a student constructs a program using our tutor is quite flexible. The tutor is offered as a web application, which further reduces the burden to use it.

Using Ask-Elle, students learning functional programming

- develop their programs incrementally,
- receive feedback about whether or not they are on the right track,
- can ask for a hint when they are stuck,
- see how a complete program is stepwise constructed.

Ask-Elle itself is implemented as a functional program, and uses fundamental concepts from software technology such as rewriting, parsing, strategies, program transformations and higher-order combinators such as the fold. The tutor is built on top of our general software framework for specifying domain reasoners, and uses the Helium compiler for Haskell. Helium gives excellent syntax-error and type-error messages, and reports dependency analysis problems in a clear way.

For any programming problem there are many solutions. Some of these solutions are syntactical variants of each other, but other solutions implement different ideas to solve a problem. A teacher can specify her exercises in Ask-Elle by giving a set of model solutions for a problem. A model solution is a program that an expert writes, using good programming practices. Our tutor supports the incremental construction, in a top-down fashion, of model solutions. It recognises incomplete versions of these solutions, together with all kinds of syntactical variants. The tutor aims to be as flexible as possible for teachers as well as for students. For example, a student may use her own names for functions and variables, and may use different, but equivalent, language constructs.

The tutor generates feedback based on a set of model solutions for a particular programming problem. A teacher can adapt feedback by annotating the model solutions. This requires translating annotated model solutions to a form which we can use to track intermediate student steps. We use programming strategies to track the intermediate steps taken by a student. A strategy, or procedure, to solve an exercise often consists of multiple steps. For example, developing an explicit recursive function on lists often consists of introducing a case distinction between the empty list and the non-empty list, and a recursive call in the non-empty list case, amongst others. A strategy may also contain a choice between different (sequences of) steps, such as either using a higher-order function, or an explicit recursive definition. Sometimes, the order in which the steps are performed is not relevant, as long as they are performed at some point.

We have developed a strategy language for describing procedures as rewrite strategies. Our strategy language is domain independent, and has been used to

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

describe strategies for exercises in mathematics, logic, and biology, in addition to programming. A strategy for a functional program describes how a student should construct a functional program for a particular problem. The basic elements of the strategy language are rewrite or refinement rules. We use these strategies to support students using our intelligent programming tutor to incrementally develop a program.

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

First-class models: On a noncausal language for higher-order and structurally dynamic modelling and simulation

GEORGE GIORGIDZE
University of Nottingham, UK

Date: Oct 2012; Advisor: Henrik Nilsson
URL: http://etheses.nottingham.ac.uk/2554/

The field of physical modelling and simulation plays a vital role in advancing numerous scientific and engineering disciplines. To cope with the increasing size and complexity of physical models, a number of modelling and simulation languages have been developed. These languages can be divided into two broad categories: causal and noncausal. Causal languages express a system model in terms of directed equations. In contrast, a noncausal model is formulated in terms of undirected equations. The fact that the causality can be left implicit makes noncausal languages more declarative and noncausal models more reusable. These are considered to be crucial advantages in many physical domains.

Current, mainstream noncausal languages do not treat equational models as first-class values; that is, a model cannot be parametrised on other models or generated at simulation runtime. This results in very limited higher-order and structurally dynamic modelling capabilities, and limits the expressiveness and applicability of noncausal languages.

This thesis is about a novel approach to the design and implementation of noncausal languages with first-class models supporting higher-order and structurally dynamic modelling. In particular, the thesis presents a language that enables: (1) higher-order modelling capabilities by embedding noncausal models as first-class entities into a functional programming language and (2) efficient simulation of noncausal models that are generated at simulation runtime by runtime symbolic processing and just-in-time compilation. These language design and implementation approaches can be applied to other noncausal languages. This thesis provides a self-contained reference for such an undertaking by defining the language semantics formally and providing an in-depth description of the implementation. The language provides noncausal modelling and simulation capabilities that go beyond the state of the art, as backed up by a range of examples presented in the thesis, and represents a significant progress in the field of physical modelling and simulation.

Concurrent pattern unification

THOMAS PAUL GIVEN-WILSON
University of Technology, Sydney, Australia

Date: August 2012; Advisor: Barry Jay
URL: http://tinyurl.com/mr5tjue

Ever since Milner showed that Church's λ-calculus can be subsumed by π-calculus, process calculi have been expected to subsume sequential computation. However, Jay and Given-Wilson show that extensional sequential computation as represented by λ-calculus is subsumed by intensional sequential computation characterised by pattern-matching as in SF-calculus. Given-Wilson, Gorla and Jay present a concurrent pattern calculus (CPC) that adapts sequential pattern-matching to symmetric pattern-unification in a process calculus. This dissertation proves that CPC subsumes both intensionality sequential computation and extensional concurrent computation, respectively SF-calculus and π-calculus, to complete a computation square. A behavioural theory is developed for CPC that is then exploited to prove that CPC is more expressive than several representative sequential and concurrent calculi. As part of its greater expressive power, CPC provides a natural language to describe interactions involving information exchange. Augmenting the pattern-matching language bondi to implement CPC yields a Concurrent bondi that is able to support web services that exploit both sequential and concurrent intensionality.

Manifest contracts

MICHAEL GREENBERG
University of Pennsylvania, USA

Date: December 2013; Advisor: Benjamin Pierce
URL: http://repository.upenn.edu/edissertations/468/

Eiffel popularized design by contract, a software design philosophy where programmers specify the requirements and guarantees of functions via executable pre- and post-conditions written in code. Findler and Felleisen brought contracts to higher-order programming, inspiring the PLT Racket implementation of contracts. Existing approaches for runtime checking lack reasoning principles and stop short of their full potential–most Racket contracts check only simple types. Moreover, the standard algorithm for higher-order contract checking can lead to unbounded space consumption and can destroy tail recursion. In this dissertation, I develop so-called manifest contract systems which integrate more coherently in the type system, and relate them to Findler-and-Felleisen-style latent contracts. I extend a manifest system with type abstraction and relational parametricity, and also show how to integrate dynamic types and contracts in a space efficient way, i.e., in a way that doesn't destroy tail recursion. I put manifest contracts on a firm type-theoretic footing, showing that they support extensions necessary for real programming. Developing these principles is the first step in designing and implementing higher-order languages with contracts and refinement types.

Type inference, Haskell and dependent types

ADAM MICHAEL GUNDRY
University of Strathclyde, UK

Date: December 2013; Advisor: Conor McBride
URL: http://tinyurl.com/kzxamvd

This thesis studies questions of type inference, unification and elaboration for languages that combine dependent type theory and functional programming. Languages such as modern Haskell have very expressive type systems, allowing the programmer a great deal of freedom. These require advanced type inference and unification algorithms to reconstruct details that were left implicit, and suitable representation of the evidence delivered by such algorithms.

The first part proposes an approach to unification and type inference, based on information increase in dependency-ordered contexts, and keeping careful track of variable scope. Two existing systems are reviewed: the Hindley-Milner type system, and units of measure in the style of Kennedy. Subtle issues relating to let-generalisation become clearer as a result. Using the same approach, an algorithm is described for Miller pattern unification in a full-spectrum dependent type theory, forming a foundation for the elaboration of dependently typed languages.

The second part introduces inch, a language that extends Haskell with type-level data and functions, and dependent product types. Type-level numbers and arithmetic operations are specifically considered, as a particularly useful source of applications, such as the perennial example of vectors (length-indexed lists). The increased expressivity in the source language is matched by a suitable core language of evidence, into which inch programs can be translated. This language is based on System FC, the existing core language used by GHC, adapted to clarify the relationships between the type and term levels. It gives a coherent operational semantics to both levels, allowing shared data and dependent functions, but retaining a clear phase distinction. The contextual approach of the first part of the thesis is used to specify the elaboration of inch into the evidence language, and applications of inch based on type-level arithmetic are demonstrated.

Types with potential: polynomial resource bounds via automatic amortized analysis

JAN HOFFMANN
Ludwig Maximilian University of Munich, Germany

Date: October 2011; Advisor: Martin Hofmann
URL: http://edoc.ub.uni-muenchen.de/13955/

A primary feature of a computer program is its quantitative performance characteristics: the amount of resources such as time, memory, and power the program needs to perform its task. Concrete resource bounds for specific hardware have many important applications in software development but their manual determination is tedious and error-prone.

This dissertation studies the problem of automatically determining concrete worst-case bounds on the quantitative resource consumption of functional programs.

Traditionally, automatic resource analyses are based on recurrence relations. The difficulty of both extracting and solving recurrence relations has led to the development of type-based resource analyses that are compositional, modular, and formally verifiable. However, existing automatic analyses based on amortization or sized types can only compute bounds that are linear in the sizes of the arguments of a function.

This work presents a novel type system that derives polynomial resource bounds from first-order functional programs. As pioneered by Hofmann and Jost for linear bounds, it relies on the potential method of amortized analysis. Types are annotated with multivariate resource polynomials, a rich class of functions that generalize non-negative linear combinations of binomial coefficients. The main theorem states that type derivations establish resource bounds that are sound with respect to the resource-consumption of programs which is formalized by a big-step operational semantics.

Simple local type rules allow for an efficient inference algorithm for the type annotations which relies on linear constraint solving only. This gives rise to an analysis system that is fully automatic if a maximal degree of the bounding polynomials is given. The analysis is generic in the resource of interest and can derive bounds on time and space usage. The bounds are naturally closed under composition and eventually summarized in closed, easily understood formulas.

The practicability of this automatic amortized analysis is verified with a publicly available implementation and a reproducible experimental evaluation. The experiments with a wide range of examples from functional programming show that the inference of the bounds only takes a couple of seconds in most cases. The derived heap-space and evaluation-step bounds are compared with the measured worst-case

behavior of the programs. Most bounds are asymptotically tight, and the constant factors are close or even identical to the optimal ones. For the first time we are able to automatically and precisely analyze the resource consumption of involved programs such as quick sort for lists of lists, longest common subsequence via dynamic programming, and multiplication of a list of matrices with different, fitting dimensions.

Compiling concurrency correctly: Verifying software transactional memory

LIYANG HU
University of Nottingham, UK

Date: June 2012; Advisor: Graham Hutton
URL: http://etheses.nottingham.ac.uk/3348/

Concurrent programming is notoriously difficult, but with multi-core processors becoming the norm, is now a reality that every programmer must face. Concurrency has traditionally been managed using low-level mutual exclusion *locks*, which are error-prone and do not naturally support the compositional style of programming that is becoming indispensable for today's large-scale software projects.

A novel, high-level approach that has emerged in recent years is that of *software transactional memory* (STM), which avoids the need for explicit locking, instead presenting the programmer with a declarative approach to concurrency. However, its implementation is much more complex and subtle, and ensuring its correctness places significant demands on the compiler writer.

This thesis considers the problem of formally verifying an implementation of STM. Utilising a minimal language incorporating only the features that we are interested in studying, we first explore various STM design choices, along with the issue of compiler correctness via the use of automated testing tools. Then we outline a new approach to concurrent compiler correctness using the notion of bisimulation, implemented using the Agda theorem prover. Finally, we show how bisimulation can be used to establish the correctness of a low-level implementation of software transactional memory.

Generic constraints for type-safe embedded programming

WILL JONES
Imperial College London, UK

Date: May 2013; Advisor: Tony Field
URL: http://pubs.doc.ic.ac.uk/will-jones-phd-thesis/

Domain-specific languages (DSLs) are everywhere, with applications in areas such as parser generation, music synthesis, parallel programming and even the design of domain-specific languages. However, while the pay-off in using a DSL may be substantial, the cost of introducing a language may be made prohibitively high by the need to construct a supporting toolchain.

A common tactic is to *embed* a DSL into a general-purpose host programming language. Existing infrastructure such as a language's compiler or type system may be re-used, provided that the embedding accurately captures the properties of the DSL. While the rich type systems and orthogonal abstraction features of modern functional languages have proved particularly capable in this regard, they are not without their shortcomings. Building type-safe functions defined over an embedded DSL can introduce application-specific type constraints that end up being imposed on the DSL data types themselves. At best, these constraints are unwieldy and at worst they can limit the range of DSL expressions that can be built.

In this thesis we tackle the problem of accurately embedding a DSL's type system into that of the purely functional language Haskell. We present a framework for expressing application-specific constraints at the point of a DSL expression's use rather than when the DSLs embedding is defined. We show how our framework can be applied more generally to capture arbitrary properties of a DSL expression and, in certain cases, how we may subsequently prove additional safety properties such as the totality of a function which operates over DSL expressions. We evaluate our techniques by illustrating their use in constructing a DSL for heterogeneous parallel programming. However, our methods have potentially wider applications such as context-dependent computation, which are also discussed.

Executable refinement types

KENNETH KNOWLES
University of California, Santa Cruz, USA

Date: March 2014; Advisor: Cormac Flanagan
URL: http://arxiv.org/abs/1403.3336

Precise specifications are integral to effective programming practice. Existing specification disciplines such as structural type systems, dynamic contracts, and extended static checking all suffer from limitations such as imprecision, false positives, false negatives, or excessive manual proof burden. New ways of expressing and enforcing program specifications are needed.

Towards that end, this dissertation introduces *executable refinement types* and establishes their metatheory and accompanying implementation techniques. Executable refinement types enrich structural type systems with basic types refined by semi-decidable predicates. Through the lens of executable refinement types, we also address the broader problem of theory and implementation for undecidable type systems.

To establish a firm foundation for the study of executable refinement types, this dissertation presents a full formal account of their metatheory. Type checking for executable refinement types is undecidable. Nonetheless, they fulfill standard metatheoretical correctness criteria including type soundness and extensional equivalence.

To perform type checking for executable refinement types we introduce *hybrid type checking*, a type enforcement strategy broadly applicable to undecidable type systems. Hybrid type checking enforces specifications via static analysis where possible and dynamic type casts where necessary. We prove that for any decidable approximation of executable refinement types, either: (1) Hybrid type checking catches some errors *statically* which the decidable approximation would miss, or (2) the decidable approximation rejects some correct program which hybrid type checking would accept.

To perform type reconstruction for executable refinement types, we radically revise the usual notion of type reconstruction. Typeability is undecidable because it subsumes type checking. Instead, we propose a more precise definition of type reconstruction as a typeability-*preserving* transformation. For decidable type systems, our definition coincides with the previous. Using our generalized notion of type reconstruction, we demonstrate that type reconstruction for executable refinement types is decidable even though type checking is not! We show this by providing a syntactic type reconstruction algorithm reminiscent of strongest postcondition calculation.

To enlarge the class of programs for which type checking is decidable, we formalize the notion of *compositional reasoning* for types systems. Because standard dependent types perform non-compositional reasoning, type checking is undecidable even when

all types appearing in a program fall in a decidable specification language. We present a variant of dependent types which uses existential types to achieve compositional reasoning. Even restricted to compositional reasoning, our type system is *exact*: It can give any term a type that completely classifies that term up to contextual equivalence. When reasoning compositionally, if all the annotations in a program fall into a decidable language, then type checking is decidable. We show this with a type checking algorithm for such programs.

Atop these theoretical foundations we implement Sage, a language blending executable refinement types, dynamic typing, and first-class types. Sages implementation includes standard type-checking machinery, compile-time computation, automatic theorem proving, dynamic contract checking, and a database of run-time failures which inform the hybrid type checker for future runs. Preliminary experiments indicate that Sage is effective at verifying many common examples statically in a reasonable amount of time. Moreover, every run-time failure in Sage can occur at most once: From then onwards it becomes a compile time failure.

Less is more: Generic programming theory and practice

JOSÉ PEDRO MAGALHÃES
Utrecht University, The Netherlands

Date: September 2012; Advisor: Johan Jeuring, Doaitse Swierstra and Andres Löh
URL: http://dspace.library.uu.nl/handle/1874/254098

Abstraction is ubiquitous in computer programming. To allow programmers to abstract from details, several programming languages have been developed. Such languages are a precise way to express commands to a computer, and eventually are translated into the only primitive operations that the computer understands. The automatic translation process from one programming language to another is the task of a compiler, itself a program which reads programs in one input language and outputs programs in another language. Compilers can be chained together, to collectively bring a program from a very high-level programming language (abstract and convenient for human reasoning) down to machine language (low-level and ready to be executed by the computer).

The work of this thesis concerns a special class of very high-level programming languages, namely statically-typed purely functional languages. In such languages, the computation to be performed is described by functions that take input and produce output. The evaluation of a program thus consists of evaluating calls to these functions. This paradigm is rather distant from the machine language at the core of the computer. The machine language deals with sequences of instructions, conditional operations, and loops. A functional language deals with function application, composition, and recursion. We choose functional programming as the starting point for our research because we believe it lends itself perfectly to express abstraction, leading to shorter and more understandable computer programs. This allows the programmer to express complex behaviour in a simple fashion, resulting in programs that are easier to adapt, compose, maintain, and reason about, all desirable properties for computer programs.

We focus on one specific form of abstraction. Computer programs manipulate data, which can either be primitive machine data (such as integer or fractional numbers) or programmer-defined data (such as lists, trees, matrices, images, etc.). There is only a small number of primitive datatypes, but a potentially infinite number of programmer-defined data. The structure of the latter data depends on the problem at hand, and while some structures appear very often (such as sequences of values), others are truly specific to a particular problem.

Some kind of functionality is generally desired for all types of data. Reading and storing files to the disk, for instance, is as important for machine integers as it is for complex healthcare databases, or genealogy trees. And not just reading and writing files: testing for equality, sorting, traversing, computing the length, all are examples of functionality that is often desired for all kinds of data. Most programming languages allow defining complex datatypes as a form of abstraction,

but few provide good support for defining behaviour that is generic over data. As such, programmers are forced to specify this behaviour over and over again, once for each new type of data, and also to adapt this code whenever the structure of their data changes. This is a tedious task, and can quickly become time-consuming, leading some programmers to write programs to generate this type of functionality automatically from the structure of data.

We think that a programming language should allow programmers to define generic programs, which specify behaviour that is generic over the type of data. Moreover, it should automatically provide generic behaviour for new data, eliminating the need for repeated writing and rewriting of trivial code that just specialises general behaviour to a particular type of data. It should do so in a convenient way for the programmer, leading to more abstract and concise programs, while remaining clear and efficient. This leads us to the two research questions we set out to answer:

1. There are many different approaches to generic programming, varying in complexity and expressiveness. How can we better understand each of the approaches, and the way they relate to each other?
2. Poor runtime efficiency, insufficient datatype support, and lack of proper language integration are often pointed out as deficiencies in generic programming implementations. How can we best address these concerns?

We answer the first question in the first part of this thesis. We start by picking a number of generic programming approaches and define a concise model for each of them. We then use this model to formally express how to embed the structural representation of data of one approach into another, allowing us to better understand the relation between different approaches. The second part of this thesis deals with answering the second question, devoting one chapter to analysing and mitigating each of the practical concerns.

Syntax and models of a non-associative composition of programs and proofs

GUILLAUME MUNCH-MACCAGNONI
Université Paris Diderot, France

Date: December 2013; Advisor: Pierre-Louis Curien
URL: http://tel.archives-ouvertes.fr/tel-00918642

The thesis is a contribution to the understanding of the nature, role, and mechanisms of polarisation in programming languages, proof theory and categorical models. Polarisation corresponds to the idea that we can relax the associativity of composition, and underlies many models of computation.

In our demonstration, we introduce *duploids*, which model polarisation directly. These compositional structures, unlike categories, do not always require composition to be associative, and they are shown to be in correspondence with adjunctions. Then, we obtain a fine-grained decomposition of continuation-passing-style translations. As an application, we explain how polarisation accounts for evaluation order in models of delimited control operators. We also show how polarisation explains constructiveness-related phenomena in proof theory, through a formulae-as-types interpretation of an involutive negation.

The cornerstone of our approach is an interactive term-based representation of proofs and programs (*L calculi*) which exposes the structure of polarities. It is based on the correspondence between abstract machines and sequent calculi, and it aims at synthesising various trends: the modelling of control, evaluation order and effects in programming languages, the quest for a relationship between categorical duality and continuations, and the interactive notion of construction in proof theory. We give a gentle introduction to our approach which only assumes elementary knowledge of simply-typed λ calculus and rewriting.

Type classes and instance chains: A relational approach

J. GARRETT MORRIS
Portland State University, USA

Date: June 2013; Advisor: Mark P. Jones
URL: http://archives.pdx.edu/ds/psu/9917

Type classes, first proposed during the design of the Haskell programming language, extend standard type systems to support overloaded functions. Since their introduction, type classes have been used to address a range of problems, from typing ordering and arithmetic operators to describing heterogeneous lists and limited subtyping. However, while type class programming is useful for a variety of practical problems, its wider use is limited by the inexpressiveness and hidden complexity of current mechanisms. We propose two improvements to existing class systems. First, we introduce several novel language features, instance chains and explicit failure, that increase the expressiveness of type classes while providing more direct expression of current idioms. To validate these features, we have built an implementation of these features, demonstrating their use in a practical setting and their integration with type reconstruction for a Hindley-Milner type system. Second, we define a set-based semantics for type classes that provides a sound basis for reasoning about type class systems, their implementations, and the meanings of programs that use them.

A quest for exactness: Program transformation for reliable real numbers

PIERRE NERON
Ecole Polytechnique, France

Date: October 2013; Advisor: Gilles Dowek and César Muñoz
URL: http://tinyurl.com/nefstw5

The notion of real number is firmly related with the notion of infinity, which is not compatible with the finiteness of computers memory. This limitation has been overcome in different ways. Since the introduction of computable numbers by Alan Turing in 1936, many representations of real and computable numbers have been studied. The most common way to deal with real numbers in programming languages is to use the floating point numbers as described in the IEEE 754 standard. For example, the 64 bits representation includes one bit for the sign, eleven for the exponent and fifty-two for the mantissa. However, this standard only represents a finite number of real numbers and therefore many rounding issues arise. In particular, none of the usual operations is always exact and therefore the result of the computation of an arithmetic expression with floating point numbers may differ from the value of this expression on real numbers.

Many techniques, from program analysis to program transformation and compilation schemes, have been developed to ensure the reliability of programs using finite representations of real numbers. However exact computations using these representation is out of scope. Exact computation may be achieved using dynamic representations such as streams of integers or polynomial representations of algebraic numbers but these representations require an unbounded amount of memory or unbounded computations.

There is at least one main reason why computations with real numbers have been so thoroughly studied. Real numbers are used to describe the physical world and many systems, namely *cyber-physical systems*, are used to control physical entities. From cars to airplanes, from medical robots to GPS chips, human develops thousands of such cyber-physical systems. Moreover, many of these systems are embedded and require a high level of safety since any failure may lead to dramatic consequences. Such critical systems often have constraints to ensure that the programs terminate and do not fail due to lack of memory, therefore exact computation techniques mentioned previously they are not suitable for use in such embedded systems.

In this thesis, we investigate a solution to the problem of computation with real numbers in embedded systems based on program transformation. The transformation we propose aims at removing square root and division operations from straight line programs (i.e. programs with no loops), such as those used in aeronautics, in order to allow exact computation over real numbers with the addition, subtraction and multiplication operations. These operations can be performed exactly in embedded programs since static analysis allows us to predict the memory required for

exact computation using a fixed point representation. This transformation does not allow to compute a real number with an arbitrary precision (the program *sqrt(2)* will still return a rounded value of $\sqrt{2}$), however it allows the system to compute exactly Boolean expressions that are built with comparisons between arithmetic expressions. Computing exactly Boolean expressions protects the control flow of the program from any rounding errors. This prevents the program effective behavior to diverge completely from its expected behavior, i.e. the one assuming the numbers are genuine real numbers. Therefore the programs produced by our transformation are somehow continuous, the effective returned value being, in the worst case, a rounding of the expected one and, if the program returns a Boolean value, then this value is exact.

This transformation algorithm relies on two fundamental algorithms. The first is a particular case of quantifier elimination on real closed fields, it eliminates square roots and divisions in Boolean expressions. The second solves a specific anti-unification problem that we called *constrained anti-unification* in order to reduce the size of the produced code. This anti-unification algorithm uses the axioms of a theory of the arithmetic and a directed acyclic graph representation in order to compute common template that allow us to optimize the size of the produced code. The constrained anti-unification algorithm is also used to extend the transformation to a richer language allowing function definitions using a *partial inlining*.

In order to still ensure the high level of safety required by the programs we are willing to transform, we also proved the correctness of this transformation in the PVS proof assistant. Indeed formal proof assistants enable the higher levels of safety and security for programs. They are used to prove properties about the behavior of these programs and ensure the correctness of such proofs more reliably than any human certification. We used the PVS proof assistant to show that the algorithm we presented not only effectively eliminates square roots and divisions but also preserves the semantics of the programs we transform. This allows us to ensure that the behavior of the transformed program is exactly the same as the expected behavior of the input program. Therefore, all the properties satisfied by this input program still hold on the transformed one.

The complete algorithm is not entirely proven in the PVS proof assistant, its correctness depends on that of the anti-unification algorithm. However, this proof is sufficient to built a proof strategy that eliminates square roots and divisions in the formulas used by the PVS proof checker. Moreover, in our transformation scheme, we provide a mechanism to generate correctness lemmas that state the semantics preservation between the variable definitions of the input and the output programs, these lemmas being quite easy to prove using the proof strategy we defined. Therefore this transformation outputs not only the transformed program but also a proof that this transformed program is semantically equivalent to the input one. Such a transformation is called a *certifying* transformation. Finally, we present how our algorithm has been able to transform a complete PVS specification of a conflict detection algorithm for air traffic management that is part of the ACCoRD system developped at NASA Langley.

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

Bidirectional data transformation by calculation

HUGO PACHECO
Universidade do Minho, Portugal

Date: September 2012; Advisor: Manuel Alcino Pereira da Cunha and José Nuno Fonseca de Oliveira
URL: http://hdl.handle.net/1822/20995

The advent of bidirectional programming, in recent years, has led to the development of a vast number of approaches from various computer science disciplines. These are often based on domain-specific languages in which a program can be read both as a forward and a backward transformation that satisfy some desirable consistency properties.

Despite the high demand and recognized potential of intrinsically bidirectional languages, they have still not matured to the point of mainstream adoption. This dissertation contemplates some usually disregarded features of bidirectional transformation languages that are vital for deployment at a larger scale. The first concerns efficiency. Most of these languages provide a rich set of primitive combinators that can be composed to build more sophisticated transformations. Although convenient, such compositional languages are plagued by inefficiency and their optimization is mandatory for a serious application. The second relates to configurability. As update translation is inherently ambiguous, users shall be allowed to control the choice of a suitable strategy. The third regards genericity. Writing a bidirectional transformation typically implies describing the concrete steps that convert values in a source schema to values a target schema, making it impractical to express very complex transformations, and practical tools shall support concise and generic coding patterns.

We first define a point-free language of bidirectional transformations (called lenses), characterized by a powerful set of algebraic laws. Then, we tailor it to consider additional parameters that describe updates, and use them to refine the behavior of intricate lenses between arbitrary data structures. On top, we propose the Multifocal framework for the evolution of XML schemas. A Multifocal program describes a generic schema-level transformation, and has a value-level semantics defined using the point-free lens language. Its optimization employs the novel algebraic lens calculus.

Static guarantees for coordinated components: A statically typed composition model for stream-processing networks

FRANK PENCZEK
University of Hertfordshire, UK

Date: September 2012; Advisor: Sven-Bodo Scholz and Clemens Grelck
URL: http://uhra.herts.ac.uk/handle/2299/9046

Does your program do what it is supposed to? Without running the program providing an answer to this question is much harder if the language does not support static type checking. Of course, even if compile-time checks are in place only certain errors will be detected: compilers can only second-guess the programmers intention. But, type based techniques go a long way in assisting programmers to detect errors in their computations earlier on.

The question if a program behaves correctly is even harder to answer if the program consists of several parts that execute concurrently and need to communicate with each other. Compilers of standard programming languages are typically unable to infer information about how the parts of a concurrent program interact with each other, especially where explicit threading or message passing techniques are used. Hence, correctness guarantees are often conspicuously absent. Concurrency management in an application is a complex problem. However, it is largely orthogonal to the actual computational functionality that a program realises. Because of this orthogonality, the problem can be considered in isolation. The largest possible separation between concurrency and functionality is achieved if a dedicated language is used for concurrency management, i.e. an additional program manages the concurrent execution and interaction of the computational tasks of the original program. Such an approach does not only help programmers to focus on the core functionality and on the exploitation of concurrency independently, it also allows for a specialised analysis mechanism geared towards concurrency-related properties.

This dissertation shows how an approach that completely decouples coordination from computation is a very supportive substrate for inferring static guarantees of the correctness of concurrent programs. Programs are described as streaming networks connecting independent components that implement the computations of the program, where the network describes the dependencies and interactions between components. A coordination program only requires an abstract notion of computation inside the components and may therefore be used as a generic and reusable design pattern for coordination. A type-based inference and checking mechanism analyses such streaming networks and provides comprehensive guarantees of the consistency and behaviour of coordination programs. Concrete implementations of components are deliberately left out of the scope of coordination programs: Components may be implemented in an external language, for example C, to provide the desired computational functionality. Based on this separation, a concise

semantic framework allows for step-wise interpretation of coordination programs without requiring concrete implementations of their components. The framework also provides clear guidance for the implementation of the language. One such implementation is presented and hands-on examples demonstrate how the language is used in practice.

---

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

Interactive functional programming

ROLAND PERERA
University of Birmingham, UK

Date: June 2013; Advisor: Paul Levy
URL: http://etheses.bham.ac.uk/4209

We outline a vision for a new kind of execution environment where applications can be debugged and re-programmed while they are being used. The overall concept we call *interactive programming*. In contrast to most other systems for live programming, interactive programming presents execution to the user as a live, explorable document. In contrast to the edit-and-continue features found in many debuggers, and to systems for patching software dynamically, we utilise a notion of *retroactive update*, where the computation transitions to a new consistent state when the program changes, rather than a hybrid of old and new. What changed in the execution is always explicit and visible to the user. Retroactive update relates our work to incremental computation.

We develop some key components of interactive programming in the setting of a pure, call-by-value functional language. We illustrate our ideas via a proof-of-concept implementation called LambdaCalc. Several important components of the overall vision, including efficient incremental update, scaling to realistic programs, supporting effectful programs, and dealing with non-termination, are left for future work. We implemented a comprehensive visualisation subsystem in LambdaCalc itself, but further performance work is required for this to be the basis of a working user interface.

Our specific achievements are as follows. First, we show how to *reify* the execution of a program into a live document which can be interactively decomposed into both sequential steps and parallel slices. We give a novel characterisation of forward and backward dynamic slicing and show that for a fixed computation, the two problems are described by a Galois connection. We extend the notion of slicing to reified computations, and formalise what it is for a slice of a computation to "explain" some part of a value. We show how being able to slice a computation interactively can help debugging.

Second, we introduce a novel execution indexing scheme which derives execution differences from program differences. Our scheme supports the wholesale reorganisation of a computation via operations such as moves and splices. The programmer is able to see the consequences of edits on the intensional structure of the execution. Where possible, node identity is preserved, allowing an edit to be made whilst an execution is being explored and the changes to be reflected in the user's current view of the execution. This allows the user to see the *impact of code changes* while debugging. We illustrate this using figures generated by our implementation. Our

self-hosted visualisation code is able to compute differences in visualisations, which we use to visualise differences in computations.

We conclude with a discussion of some of the challenges facing the proposed paradigm: space requirements, visualising large computations and data structures, computational effects, and integrating with environments that lack support for retroactive update.

---

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

Multi-parameter and optional type classes for Haskell

RODRIGO GERALDO RIBEIRO
Federal University of Minas Gerais, Brazil

Date: July 2013; Advisor: Carlos Camarão de Figueiredo and Lucília Camarão de Figueiredo
URL: http://tinyurl.com/n5d4got

The introduction of multi-parameter type classes in Haskell — instead of just in the language's mostly used compiler (the Glasgow Haskell Compiler), as occurs nowadays — has been delayed due to problems related to ambiguity (which occur due to the lack of type specialization during type inference) and to the existence of distinct language extensions to handle the problem, namely functional dependencies and type families.

This thesis discusses the problem of ambiguity in Haskell and proposes a type system for Haskell that supports the definition of multi-parameter type classes without the need of any extensions.

An alternative definition of ambiguity is proposed for Haskell, where the existence of more than one instance (i.e. more than one type derivation) for the same type of an expression is considered only when overloading is, or should have been, resolved. We identify this condition by the presence of unreachable variables in constraints on the type of the expression (the condition that nowadays characterizes ambiguity in Haskell). Ambiguity becomes then an error that arises due to the existence of two or more instances but only for expressions for which there is no context in which they could be placed that would allow a selection between one of these instances. This is in full agreement with Haskell's context-dependent overloading policy, where overloading resolution depends not only on the types of expressions but also on the context of occurrence.

Adopting our approach to ambiguity detection in Haskell would eliminate the need of using functional dependencies or type families for the specific purpose of dealing with ambiguity. It would also enable Haskell compilers to provide more helpful ambiguity-related error messages.

Algorithms for constraint set satisfiability and simplification of Haskell type class constraints are used during type inference in order to allow the inference of more accurate types and to detect ambiguity. Both constraint set satisfiability and simplification are in general undecidable, and the use of these algorithms may cause non-termination of type inference. The thesis presents algorithms for these problems that terminate on any given input, based on the use of a criterion that is tested on each recursive step of the constraint-set satisfiability algorithm. The use of this criterion eliminates the need of imposing syntactic conditions on Haskell type class and instance declarations in order to guarantee termination of type inference in the presence of multi-parameter type classes, and allows program compilation without the need of compiler flags for lifting such restrictions. Undecidability of the problems implies the existence of instances for which the algorithm incorrectly

reports unsatisfiability, but we are not aware of any practical example where this occurs.

The definition of overloaded symbols is also allowed without the need of specifying a type class. In this case, the least generalization of the types of available definitions, computed by a simple anti-unification algorithm, is used as the overloaded symbol's type.

A type inference algorithm that is sound and complete with respect to the proposed type system (and the revised definition of constraint-set satisfiability, that makes it a decidable relation) is presented and implemented.

(Dissertation language: Portuguese)

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming

TIARK ROMPF
École Polytechnique Fédérale de Lausanne, Switzerland

Date: July 2012; Advisor: Martin Odersky
URL: http://infoscience.epfl.ch/record/180642

Programs expressed in a high-level programming language need to be translated to a low-level machine dialect for execution. This translation is usually accomplished by a compiler, which is able to translate any legal program to equivalent low-level code. But for individual source programs, automatic translation does not always deliver good results: Software engineering practice demands generalization and abstraction, whereas high performance demands specialization and concretization. These goals are at odds, and compilers can only rarely translate expressive high-level programs to modern hardware platforms in a way that makes best use of the available resources.

Explicit program generation is a promising alternative to fully automatic translation. Instead of writing down the program and relying on a compiler for translation, developers write a program generator, which produces a specialized, efficient, low-level program as its output. However, developing high-quality program generators requires a very large effort that is often hard to amortize.

In this thesis, we propose a hybrid design: Integrate compilers into programs so that programs can take control of the translation process, but rely on libraries of common compiler functionality for help.

We present Lightweight Modular Staging (LMS), a generative programming approach that lowers the development effort significantly. LMS combines program generator logic with the generated code in a single program, using only types to distinguish the two stages of execution. Through extensive use of component technology, LMS makes a reusable and extensible compiler framework available at the library level, allowing programmers to tightly integrate domain-specific abstractions and optimizations into the generation process, with common generic optimizations provided by the framework. Compared to previous work on program generation, a key aspect of our design is the use of staging not only as a front-end, but also as a way to implement internal compiler passes and optimizations, many of which can be combined into powerful joint simplification passes.

LMS is well suited to develop embedded domain specific languages (DSLs) and has been used to develop powerful performance-oriented DSLs for demanding domains such as machine learning, with code generation for heterogeneous platforms including GPUs. LMS has also been used to generate SQL for embedded database queries and JavaScript for web applications.

Trace-based just-in-time compilation for lazy functional programming languages

THOMAS SCHILLING
University of Kent, UK

Date: April 2013; Advisor: Simon Thompson
URL: http://tinyurl.com/p6nosau

This thesis investigates the viability of trace-based just-in-time (JIT) compilation for optimising programs written in the lazy functional programming language Haskell. A trace-based JIT compiler optimises only execution paths through the program, which is in contrast to method-based compilers that optimise complete functions at a time. The potential advantages of this approach are shorter compilation times and more natural interprocedural optimisation.

Trace-based JIT compilers have previously been used successfully to optimise programs written in dynamically typed languages such as JavaScript, Python, or Lua, but also statically typed languages like Java or the Common Language Runtime (CLR). Lazy evaluation poses implementation challenges similar to those of dynamic languages, so trace-based JIT compilation promises to be a viable approach. In this thesis we focus on program performance, but having a JIT compiler available can simplify the implementation of features like runtime inspection and mobile code.

We implemented Lambdachine, a trace-based JIT compiler which implements most of the pure subset of Haskell. We evaluate Lambdachines performance using a set of micro-benchmarks and a set of larger benchmarks from the "spectral" category of the Nofib benchmark suite. Lambdachines performance (excluding garbage collection overheads) is generally about 10 to 20 percent slower than GHC on statically optimised code. We identify the two main causes for this slow-down: trace selection and impeded heap allocation optimisations due to unnecessary thunk updates.

Towards safe and efficient functional reactive programming

NEIL SCULTHORPE
University of Nottingham, UK

Date: July 2011; Advisor: Henrik Nilsson
URL: http://etheses.nottingham.ac.uk/1981/1/thesis.pdf

Functional Reactive Programming (FRP) is an approach to reactive programming where systems are structured as networks of functions operating on time-varying values (signals). FRP is based on the synchronous data-flow paradigm and supports both continuous-time and discrete-time signals (hybrid systems). What sets FRP apart from most other reactive languages is its support for systems with highly dynamic structure (dynamism) and higher-order reactive constructs (higher-order data-flow). However, the price paid for these features has been the loss of the safety and performance guarantees provided by other, less expressive, reactive languages.

Statically guaranteeing safety properties of programs is an attractive proposition. This is true in particular for typical application domains for reactive programming such as embedded systems. To that end, many existing reactive languages have type systems or other static checks that guarantee domain-specific constraints, such as feedback being well-formed (causality analysis). However, compared with FRP, they are limited in their capacity to support dynamism and higher-order data-flow. On the other hand, as established static techniques do not suffice for highly structurally dynamic systems, FRP generally enforces few domain-specific constraints, leaving the FRP programmer to manually check that the constraints are respected. Thus, there is currently a trade-off between static guarantees and dynamism among reactive languages.

This thesis contributes towards advancing the safety and efficiency of FRP by studying highly structurally dynamic networks of functions operating on mixed (yet distinct) continuous-time and discrete-time signals. First, an ideal denotational semantics is defined for this kind of FRP, along with a type system that captures domain-specific constraints. The correctness and practicality of the language and type system are then demonstrated by proof-of-concept implementations in Agda and Haskell. Finally, temporal properties of signals and of functions on signals are expressed using techniques from temporal logic, as motivation and justification for a range of optimisations.

Amortised resource analysis for lazy functional programs

HUGO MIGUEL OLIVEIRA ROMUALDO SIMÕES
Universidade do Porto, Portugal

Date: February 2014; Advisor: António Mário da Silva Marcos Florido and Kevin Hammond
URL: http://hdl.handle.net/10216/71806

This thesis describes the first successful attempt, of which we are aware, to define an automatic, type-based static analysis of resource bounds for lazy functional programs. Lazy evaluation allows improved modularity of programs, but often makes resource usage difficult to predict. Our analysis uses the automatic amortisation approach developed by Hofmann and Jost, which was previously restricted to eager evaluation. In this thesis, we extend this work to a lazy setting by capturing the costs of unevaluated expressions in type annotations and by amortising the payment of these costs using a notion of *lazy potential*. We present our analysis as a proof system for predicting (at compile-time) total heap allocations of a minimal functional language (including higher-order functions and recursive data types) and define a formal cost model based on Launchbury?s natural semantics for lazy evaluation. We prove the soundness of our analysis with respect to the cost model. Our approach is illustrated by type derivations of a number of representative and non-trivial examples that have been analysed using a prototype implementation of our analysis.

Lightweight verification of functional programs

NICHOLAS SMALLBONE
Chalmers University of Technology, Sweden

Date: May 2013; Advisor: Koen Claessen
URL: http://tinyurl.com/owne3uk

We have built several tools to help with testing and verifying functional programs. All three tools are based on QuickCheck properties. Our goal is to allow programmers to do more with QuickCheck properties than just test them.

The first tool is QuickSpec, which finds equational specifications, and can be used to help with writing a specification or for program understanding. On top of QuickSpec, we have built HipSpec, which proves properties about Haskell programs, and uses QuickSpec to find the necessary lemmas. We also describe PULSE and eqc_par_statem, which together can be used to find race conditions in Erlang programs.

We believe that testable properties are a good basis for reasoning and verification, and that they give many of the benefits of formal verification without the cost of proof. The chief reason is that they are formal specifications for which the programmer can always get a counterexample when they are false. Furthermore, using testable properties allows us to write better tools. None of our tools would be possible if our properties were not testable.

We also present work on encoding types in first-order logic, an essential component when using first-order provers to reason about programs. Our encodings are simple but extremely efficient, as evidenced by benchmarks. We develop the theory behind sound type encodings, and have written tools that implement our ideas.

Reliable massively parallel symbolic computing: Fault tolerance for a distributed Haskell

ROBERT STEWART
Heriot-Watt University, UK

Date: November 2013; Advisor: Phil Trinder and Patrick Maier
URL: http://tinyurl.com/kq6hbw2

As the number of cores in manycore systems grows exponentially, the number of failures is also predicted to grow exponentially. Hence massively parallel computations must be able to tolerate faults. Moreover new approaches to language design and system architecture are needed to address the resilience of massively parallel heterogeneous architectures.

Symbolic computation has underpinned key advances in Mathematics and Computer Science, for example in number theory, cryptography, and coding theory. Computer algebra software systems facilitate symbolic mathematics. Developing these at scale has its own distinctive set of challenges, as symbolic algorithms tend to employ complex irregular data and control structures. SymGridParII is a middleware for parallel symbolic computing on massively parallel High Performance Computing platforms. A key element of SymGridParII is a domain specific language (DSL) called Haskell Distributed Parallel Haskell (HdpH). It is explicitly designed for scalable distributed-memory parallelism, and employs work stealing to load balance dynamically generated irregular task sizes.

To investigate providing scalable fault tolerant symbolic computation we design, implement and evaluate a reliable version of HdpH, HdpH-RS. Its reliable scheduler detects and handles faults, using task replication as a key recovery strategy. The scheduler supports load balancing with a fault tolerant work stealing protocol. The reliable scheduler is invoked with two fault tolerance primitives for implicit and explicit work placement, and 10 fault tolerant parallel skeletons that encapsulate common parallel programming patterns. The user is oblivious to many failures, they are instead handled by the scheduler.

An operational semantics describes small-step reductions on states. A simple abstract machine for scheduling transitions and task evaluation is presented. It defines the semantics of supervised futures, and the transition rules for recovering tasks in the presence of failure. The transition rules are demonstrated with a fault-free execution, and three executions that recover from faults.

The fault tolerant work stealing has been abstracted in to a Promela model. The SPIN model checker is used to exhaustively search the intersection of states in this automaton to validate a key resiliency property of the protocol. It asserts that an initially empty supervised future on the supervisor node will eventually be full in the presence of all possible combinations of failures.

The performance of HdpH-RS is measured using five benchmarks. Supervised scheduling achieves a speedup of 757 with explicit task placement and 340 with

lazy work stealing when executing Summatory Liouville up to 1400 cores of a HPC architecture. Moreover, supervision overheads are consistently low scaling up to 1400 cores. Low recovery overheads are observed in the presence of frequent failure when lazy on-demand work stealing is used. A Chaos Monkey mechanism has been developed for stress testing resiliency with random failure combinations. All unit tests pass in the presence of random failure, terminating with the expected results.

https://doi.org/10.1017/S0956796814000215 Published online by Cambridge University Press

Practical programming with substructural types

JESSE A. TOV
Northeastern University, USA

Date: February 2012; Advisor: Riccardo Pucella
URL: http://iris.lib.neu.edu/comp_sci_diss/35/

Substructural logics remove from classical logic rules for reordering, duplication, or dropping of assumptions. Because propositions in such a logic may no longer be freely copied or ignored, this suggests understanding propositions in substructural logics as representing resources rather than truth. For the programming language designer, substructural logics thus provide a framework for considering type systems that can track the changing states of logical and physical resources.

While several substructural type systems have been proposed and implemented, many of these have targeted substructural types at a particular purpose, rather than offering them as a general facility. The more general substructural type systems have been theoretical in nature and too unwieldy for practical use. This dissertation presents the design of a general purpose language with substructural types, and discusses several language design problems that had to be solved in order to make substructural types useful in practice.

First class syntax, semantics, and their composition

MARCOS VIERA
Utrecht University, The Netherlands
PEDECIBA-Universidad de la República, Uruguay

Date: March 2013; Advisor: Doaitse Swierstra and Alberto Pardo
URL: http://dspace.library.uu.nl/handle/1874/269786

Ideally complexity is managed by composing a system out of quite a few, more or less independent, and much smaller descriptions of various aspects of the overall artifact. When describing (extensible) programming languages, attribute grammars have turned out to be an excellent tool for modular definition and integration of their different aspects.

In this thesis we show how to construct a programming language implementation by composing a collection of attribute grammar fragments describing separate aspects of the language. More specifically we describe a coherent set of libraries and tools which together make this possible in Haskell, *where the correctness of the composition is enforced through the Haskell type system*'s ability to represent attribute grammars as plain Haskell values and their interfaces as Haskell types.

Semantic objects thus constructed can be combined with parsers which are constructed on the fly out of parser fragments and are also represented as typed Haskell values. Again the type checker prevents insane compositions.

As a small case study of the techniques proposed in this thesis, we implemented a compiler for the (Pascal-like) imperative language Oberon0. Through an incremental design, we show the modularity capacities of our techniques.

Functional query languages with categorical types

RYAN WISNESKY
Harvard University, USA

Date: November 2013; Advisor: Greg Morrisett
URL: http://wisnesky.net/dissertation.pdf

We study three category-theoretic types in the context of functional query languages (typed lambda-calculi extended with additional operations for bulk data processing). The types we study are:

- *The dependent identity type.* By adding identity types to the simply-typed lambda-calculus we obtain a language where embedded dependencies are first-class objects that can be manipulated by the programmer and used for optimization. We prove that the chase re-writing procedure is sound for this language.
- *The type of propositions.* By adding propositions to the simply-typed lambda-calculus, we obtain higher-order logic. We prove that every hereditarily domain-independent higher-order logic program can be translated into the nested relational algebra, thereby allowing higher-order logic to be used as a query language and giving a higher-order generalization of Codd's theorem.
- *The type of finitely presented categories.* By adding types for finitely presented categories to the simply-typed lambda-calculus we obtain a schema mapping language for the functorial data model. We define FQL, the first query language for this data model, investigate its metatheory, and build a SQL compiler for FQL.

Theory and implementation of coercive subtyping

TAO XUE
Royal Holloway, University of London, UK

Date: February 2013; Advisor: Zhaohui Luo
URL: http://tinyurl.com/nrxgc3y

Coercive subtyping is a useful and powerful framework of subtyping for type theories. The thesis is based on Luos study of coercive subtyping and consists of the following parts.

We point out the problem in the old formulation of coercive subtyping introduced by Luo. Luos previous work was based on a notion of basic subtyping rules which turns out to be unnecessarily general. It may lead the type system with coercive subtyping inconsistent. We give a new and adequate formulation $T[C]$, the system that extends a type theory T with coercive subtyping based on a set C of basic subtyping judgements. Our new formulation fix the problem of previous study.

We study the relation between a type system and its coercive subtyping extension. We naturally think the extension with coercive subtyping should be a conservative extension, since we consider coercion as an abbreviation mechanism which should not increase the power of the system. But we find that the traditional notion of conservative extension is not enough to describe the relation. We employ the idea of definitional extension from mathematical logic and formulate it in type theory. Some intermediate systems are introduced to help our study: the star-calculus $T[C]^*$, in which the positions that require coercion insertions are marked; system $T[C]_{0K}$ which does not contain the coercion application rules. We show that $T[C]$ is equivalent to $T[C]^*$, $T[C]^*$ is a definitional extension of $T[C]_{0K}$, and $T[C]_{0K}$ is a conservative extension of T. This makes clear what we mean by coercive subtyping being a conservative extension and amends a technical problem that has led to a gap in the earlier conservativity proof.

Another part of the work in this thesis concerns the implementation of coercive subtyping in the proof assistant Plastic. Coercive subtyping was implemented in Plastic by Paul Callaghan. We have done some improvement based on that work, fixed some problems of Plastic, and implemented a new kind of data type called dot-types, which are special data types useful in formal semantics to describe interesting linguistic phenomena such as copredication, in Plastic.