JFP **28**, e17, 23 pages, 2018.
doi:10.1017/S0956796818000175

PhD Abstracts

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

Many students complete PhDs in functional programming each year. As a service to the community, the Journal of Functional Programming publishes the abstracts from PhD dissertations completed during the previous year.

The abstracts are made freely available on the JFP website, i.e. not behind any paywall. They do not require any transfer of copyright, merely a license from the author. A dissertation is eligible for inclusion if parts of it have or could have appeared in JFP, that is, if it is in the general area of functional programming. The abstracts are not reviewed.

We are delighted to publish 17 abstracts in this round and hope that JFP readers will find many interesting dissertations in this collection that they may not otherwise have seen. If a student or advisor would like to submit a dissertation abstract for publication in this series, please contact the series editor for further details.

Graham Hutton
PhD Abstract Editor

Integration of the Process Algebra CSP in Dependent Type Theory – Formalisation and Verification

BASHAR IGRIED DEB ALKHAWALDEH
Swansea University, UK

Date: February 2018; Advisor: Anton Setzer and Ulrich Berger
URL: https://tinyurl.com/yagoeb7y

We introduce a library called CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. We will enhance processes by a monad structure. The monad structure facilitates combining processes in a modular way, and allows to define recursion as a direct operation on processes. Processes are defined coinductively as non-well-founded trees. The nodes of the tree are formed by a an atomic one step relation, which determines for a process the external, internal choices, and termination events it can choose, and whether the process has terminated. The data type of processes is inspired by Setzer and Hancock's notion of interactive programs in dependent type theory. The operators of CSP will be defined rather than atomic operations, and compute new elements of the data type of processes from existing ones.

The approach will make use of advanced type theoretic features: the use of inductive-recursively defined universes; the definition of coinductive types by their observations, which has similarities to the notion of an object in object-oriented programming; the use of sized types for coinductive types, which allow coinductive definitions in a modular way; the handling of finitary information (names of processes) in a coinductive settings; the use of named types for automatic inference of arguments similar to its use in template Meta-programming in C++; and the use of interactive programs in dependent type theory.

We introduce a simulator as an interactive program in Agda. The simulator allows to observe the evolving of processes following external or internal choices. Our aim is to use this in order to simulate railway interlocking system and write programs in Agda which directly use CSP processes.

Then we extend the trace semantics of CSP to the monadic setting. We implement this semantics, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. Because of the monadic settings, some adjustments need to be made to these laws.

Next we implement the more advanced semantics of CSP, the stable failures semantics and the failures divergences infinite traces semantics (FDI), in CSP-Agda, and define the corresponding refinement and equality relations. Direct proofs in these semantics are cumbersome, and we develop a technique of showing algebraic laws in those semantics in an indirect way, which is much easier. We introduce divergence-respecting weak bisimilarity and strong bisimilarity in CSP-Agda, and

show that both imply equivalence with respect to stable failures and FDI semantics. Now we show certain algebraic laws with respect to one of these two bisimilarity relations. As a case study, we model and verify a possible scenario for railways in CSP-Agda and in standard CSP tools.

---

https://doi.org/10.1017/S0956796818000175 Published online by Cambridge University Press

Distributive Interaction of Algebraic Effects

KWOK-HO CHEUNG
University of Oxford, UK

Date: February 2018; Advisor: Jeremy Gibbons and Faris Abou-Saleh
URL: https://tinyurl.com/ycgykcuo

While monadic effects—that is, the view of computational effects as categorical monads—are widespread in modern functional programming, the idea that effects can be formulated equivalently as algebraic theories seems a less familiar one to programmers. Concretely, an algebraic theory consists of two parts—a set of operations from which one is able to construct syntactic terms, and equational laws, each of which identify two particular terms. For example, the Monoid type class in Haskell describes a theory, for it consists of two operations one might refer to as multiplication and unit, with implicit associativity and unit laws matching the same concept of monoid in algebra. One is then able to instantiate types that admit such structure of a unital multiplication, which turn out to be very pervasive in programming. Indeed, the canonical instance (corresponding exactly to the free models of the theory of monoids) is the list monad. An appeal of this alternate view of computational effect—also known as algebraic effect—is the clear decoupling between specification and implementation (or in more model-theoretic terms, syntax versus semantics). With monads, this distinction is arguably less clear.

But perhaps the most compelling reason for considering algebraic effects is the relative ease by which such effects can be combined. This point is clearly of much relevance to the semantics of programming languages since in much of modern software development, one often deals with multiple interacting computational effects. As a simple example, we may want a program that not only keeps track of some state across the computation (e.g. a parser consuming a string of text), but also account for the possibility of failure, perhaps even with multiple such exceptional values. In Haskell, where such effects are explicit in the (monadic) types, we can express the combination of "statefulness" and "exceptionality" as itself a monadic type (whether directly, or by using monad transformers—a popular technique for combining monads in Haskell). But it is well known that there is more than one way to combine these two effects, each corresponding to a different composition of the underlying functors. One choice of composition reverts the state to its original value in the event of a failure, whereas another choice of composition does not. Neither of the interactions can be considered canonical, since both have their use cases. It turns out that under the lens of algebraic effects, both interactions can be understood in terms of very straightforward amalgamations of the respective equational laws. Specifically, the latter is given by taking a simple union of the two theories, while the former in addition demands equations for commutativity between each pair of stateful and exceptional operations. Both of these algebraic

constructions arise naturally from the categorical structure of *Lawvere theories*—the more abstract formulation of equational theories, as categories.

In this dissertation we seek to further the understanding of combining effects, especially from this more algebraic perspective. Of particular interest is a *distributive tensor* construction on Lawvere theories that does not seem to be very widely known. Similar to the above, this takes a simple union of two theories, but demands additional equations for *distributivity* of operations in one theory over those of the other (e.g. in the same sense that multiplication distributes over addition in any ring-like algebraic structure). There are some clear parallels between this notion and distributive laws of monads (that underlie several of the most common monad transformers). We make some steps towards establishing a more precise relationship, giving examples where the two notions coincide. The distributive tensor plays a leading role in many examples of computational interest—two such applications will be considered here in some depth.

From the observation that various combinatorial search strategies are characterised by two equivalent formulations—as *bunch* monadic types, and as more structured theories of monoids—we give a number of correspondence results. In particular, the bunch type describing a kind of depth-bounded traversal is shown to be models of a distributive tensor. We also consider the free models of this theory, giving rise to a monad, and by imposing symmetry on the monoid operation, obtain another distributive tensor theory matching closely breadth-first traversal. Depth-first traversal is implicit in the list monad, and it is shown that the list monad transformer is exactly a distributive tensor from the theory of monoids. This is in contrast to the equational presentation of the alternative "done right" list transformer, which is clarified: while it too exhibits distributivity of the monoid operation, it does so crucially only in the leftwards direction and not the right.

The second application considers in some detail a derivation of the geometrically convex monad—the combination of probabilistic and nondeterministic choice effects, called *combined choice*—in a relatively simpler setting than the usual (domain-theoretic) presentation. As such, its characterisation as a distributive tensor is clearer. The results building up to this are then applied to equational reasoning. Under this lens, one is more able to identify an incorrect assumption in various equational axiomatisations of effects (such as probabilistic choice) overlooked in the literature. Specifically, it was thought that the monadic bind operation distributes over probabilistic choice, both in the leftward and rightward directions. It turns out the rightward direction is a far more contentious property, particularly if one assumes that such properties carry over without fuss when combined with other effects. For example, in combined choice we show that rightwards distributivity of bind over probabilistic choice effectively leads to the collapse of the probabilistic operations.

As it can be difficult to get the equational properties of the interaction between probabilistic and nondeterministic choice right, a technique is explored for reasoning about such equations visually, by taking a geometric interpretation of the free models of combined choice, as convex polygons on a plane. We illustrate the use of this diagrammatic model by exhibiting non-distributivity of nondeterministic over probabilistic choice.

https://doi.org/10.1017/S0956796818000175 Published online by Cambridge University Press

Adding Dependent Types to Class-Based Mutable Objects

JOANA CORREIA CAMPOS
University of Lisbon, Portugal

Date: January 2018; Advisor: Vasco Manuel Thudichum de Serpa Vasconcelos
URL: https://tinyurl.com/ybx4aba5

In this thesis, we present an imperative object-oriented language featuring a dependent type system designed to support class-based programming and inheritance. The system brings classes and dependent types into play so as to enable types (classes) to be refined by value parameters (indices) drawn from some constraint domain. This combination allows statically checking interesting properties of imperative programs that are impossible to check in conventional static type systems for objects.

From a pragmatic point of view, this work opens the possibility to combine the scalability and modularity of object orientation with the safety provided by dependent types in the form of index refinements. These may be used to provide additional guarantees about the fields of objects, and to prevent, for example, a method call that could leave an object in a state that would violate the class invariant. One key feature is that the programmer is not required to prove equations between indices issued by types, but instead the typechecker depends on external constraint solving. From a theoretic perspective, our fundamental contribution is to formulate a system that unifies the three very different features: dependent types, mutable objects and class-based inheritance with subtyping. Our approach includes universal and existential types, as well as union types. Subtyping is induced by inheritance and quantifier instantiation. Moreover, dependent types require the system to track type varying objects, a feature missing from standard type systems in which the type is constant throughout the object's lifetime. To ensure that an object is used correctly, aliasing is handled via a linear type discipline that enforces unique references to type varying objects. The system is decidable, provided indices are drawn from some decidable theory, and proved sound via subject reduction and progress. We also formulate a typechecking algorithm that gives a precise account of quantifier instantiation in a bidirectional style, combining type synthesis with checking. We prove that our algorithm is sound and complete.

By way of example, we implement insertion and deletion for binary search trees in an imperative style, and come up with types that ensure the binary search tree invariant. To attest the relevance of the language proposed, we provide a fully functional prototype where this and other examples can be typechecked, compiled and run. The prototype can be found at http://rss.di.fc.ul.pt/tools/dol/.

The Remote Monad

JUSTIN DAWSON
University of Kansas, USA

Date: May 2018; Advisor: Andrew Gill
URL: https://tinyurl.com/ybyzjx7r

Remote Procedure Calls are an integral part of the internet of things and cloud computing. However, remote procedures, by their very nature, have an expensive overhead cost of a network round trip. There have been many optimizations to amortize the network overhead cost, including asynchronous remote calls and batching requests together.

In this dissertation, we present a principled way to batch procedure calls together, called the Remote Monad. The support for monadic structures in languages such as Haskell can be utilized to build a staging mechanism for chains of remote procedures. Our specific formulation of remote monads uses natural transformations to make modular and composable network stacks which can automatically bundle requests into packets by breaking up monadic actions into ideal packets. By observing the properties of these primitive operations, we can leverage a number of tactics to maximize the size of the packets.

We have created a framework which has been successfully used to implement the industry standard JSON-RPC protocol, a graphical browser-based library, an efficient byte string implementation, a library to communicate with an Arduino board and database queries all of which have automatic bundling enabled. We demonstrate that the result of this investigation is that the cost of implementing bundling for remote monads can be amortized almost for free, when given a user-supplied packet transportation mechanism.

Proofs and Programs about Open Terms

FRANCISCO FERREIRA RUIZ
McGill University, Canada

Date: April 2018; Advisor: Brigitte Pientka
URL: https://tinyurl.com/y9vu7sz8

Formal deductive systems are very common in computer science. They are used to represent logics, programming languages, and security systems. Moreover, writing programs that manipulate them and that reason about them is important and common. Consider proof assistants, language interpreters, compilers and other software that process input described by formal systems. This thesis shows that contextual types can be used to build tools for convenient implementation and reasoning about deductive systems with binders. We discuss three aspects of this: the reconstruction of implicit parameters that makes writing proofs and programs with dependent types easier, the addition of contextual objects to an existing programming language that make implementing formal systems with binders easier, and finally, we explore the idea of embedding the logical framework LF using contextual types in fully dependently typed theory. These are three aspects of the same message: programming using the right abstraction allows us to solve deeper problems with less effort. In this sense we want: easier to write programs and proofs (with implicit parameters), languages that support binders (by embedding a syntactic framework using contextual types), and the power of the logical framework LF with the expressivity of dependent types.

Conversational Concurrency

TONY GARNOCK-JONES
Northeastern University, USA

Date: December 2017; Advisor: Matthias Felleisen
URL: https://tinyurl.com/yax65dzv

Concurrent computations resemble conversations. In a conversation, participants direct utterances at others and, as the conversation evolves, exploit the known common context to advance the conversation. Similarly, collaborating software components share knowledge with each other in order to make progress as a group towards a common goal.

This dissertation studies concurrency from the perspective of cooperative knowledge-sharing, taking the conversational exchange of knowledge as a central concern in the design of concurrent programming languages. In doing so, it makes five contributions:

1. It develops the idea of a common *dataspace* as a medium for knowledge exchange among concurrent components, enabling a new approach to concurrent programming.
   While dataspaces loosely resemble both "fact spaces" from the world of Linda-style languages and Erlang's collaborative model, they significantly differ in many details.
2. It offers the first crisp formulation of cooperative, conversational knowledge-exchange as a mathematical model.
3. It describes two faithful implementations of the model for two quite different languages.
4. It proposes a completely novel suite of linguistic constructs for organizing the internal structure of individual actors in a conversational setting.
   The combination of dataspaces with these constructs is dubbed Syndicate.
5. It presents and analyzes evidence suggesting that the proposed techniques and constructs combine to simplify concurrent programming.

The dataspace concept stands alone in its focus on representation and manipulation of conversational frames and conversational state and in its integral use of explicit epistemic knowledge. The design is particularly suited to integration of general-purpose I/O with otherwise-functional languages, but also applies to actor-like settings more generally.

Domain Specific Languages for Small Embedded Systems

MARK DOUGLAS GREBE
University of Kansas, USA

Date: May 2018; Advisor: Andrew Gill
URL: https://tinyurl.com/yd53hale

Resource limited embedded systems provide a great challenge to programming using functional languages. Although these embedded systems cannot be programmed directly with Haskell, I show that an embedded domain specific language is able to be used to program them, and provides a user friendly environment for both prototyping and full development. The Arduino line of microcontroller boards provide a versatile, low cost and popular platform for development of these resource limited systems, and I use these boards as the platform for my DSL research.

First, I provide a shallowly embedded domain specific language, and a firmware interpreter, allowing the user to program the Arduino while tethered to a host computer. Shallow EDSLs allow a programmer to program using many of the features of a host language and its syntax, but sacrifice performance. Next, I add a deeply embedded version, allowing the interpreter to run standalone from the host computer, as well as allowing the code to be compiled to C and then machine code for efficient operation. Deep EDSLs provide better performance and flexibility, through the ability to manipulate the abstract syntax tree of the DSL program, but sacrifice syntactical similarity to the host language. Using Haskino, my EDSL designed for Arduino microcontrollers, and a compiler plugin for the Haskell GHC compiler, I show a method for combining the best aspects of shallow and deep EDSLs. The programmer is able to write in the shallow EDSL, and have it automatically transformed into the deep EDSL. This allows the EDSL user to benefit from powerful aspects of the host language, Haskell, while meeting the demanding resource constraints of the small embedded processing environment.

The Worker-Wrapper Transformation: Getting it Right and Making it Better

JENNIFER HACKETT
University of Nottingham, UK

Date: December 2017; Advisor: Graham Hutton
URL: https://tinyurl.com/yad3b9jz

A program optimisation must have two key properties: it must preserve the meaning of programs (correctness) while also making them more efficient (improvement). An optimisation's correctness can often be rigorously proven using formal mathematical methods, but improvement is generally considered harder to prove formally and is thus typically demonstrated with empirical techniques such as benchmarking. The result is a conspicuous "reasoning gap" between correctness and efficiency.

In this thesis, we focus on a general-purpose optimisation: the worker/wrapper transformation. We develop a range of theories for establishing correctness and improvement properties of this transformation that all share a common structure. Our development culminates in a single theory that can be used to reason about both correctness and efficiency in a unified manner, thereby bridging the reasoning gap.

Design and Implementation of the Futhark Programming Language

TROELS HENRIKSEN
University of Copenhagen, Denmark

Date: November 2017; Advisor: Fritz Henglein and Cosmin Eugen Oancea
URL: https://tinyurl.com/y9buptxv

In this thesis we describe the design and implementation of Futhark, a small data-parallel purely functional array language that offers a machine-neutral programming model, and an optimising compiler that generates efficient OpenCL code for GPUs. The overall philosophy is based on seeking a middle ground between functional and imperative approaches. The specific contributions are as follows:

First, we present a *moderate flattening* transformation aimed at enhancing the degree of parallelism, which is capable of exploiting easily accessible parallelism. Excess parallelism is efficiently sequentialised, while keeping access patterns intact, which then permits further locality-of-reference optimisations. We demonstrate this capability by showing instances of automatic loop tiling, as well as optimising memory access patterns.

Second, to support the flattening transformation, we present a lightweight system of size-dependent types that enables the compiler to reason symbolically about the size of arrays in the program, and that reuses general-purpose compiler optimisations to infer relationships between sizes.

Third, we furnish Futhark with novel parallel combinators capable of expressing efficient sequentialisation of excess parallelism, as well as their fusion rules.

Fourth, in order to express efficient programmer-written sequential code inside parallel constructs, we introduce support for safe in-place updates, with type system support to ensure referential transparency and equational reasoning.

Fifth, we perform an evaluation on 21 benchmarks that demonstrates the impact of the language and compiler features, and shows application-level performance that is in many cases competitive with hand-written GPU code.

Sixth, we make the Futhark compiler freely available with full source code and documentation, to serve both as a basis for further research into functional array programming, and as a useful tool for parallel programming in practice.

Extensions to Type Classes and Pattern Match Checking

GEORGIOS KARACHALIAS
KU Leuven, Belgium

Date: May 2018; Advisor: Tom Schrijvers
URL: https://tinyurl.com/y8o3c5dz

Static typing is one of the most prominent techniques in the design of programming languages for making software more safe and more reusable. Furthermore, it provides opportunities for compilers to automatically generate boilerplate code, using mathematical foundations. In this thesis, we extend upon the design of Haskell, a general-purpose functional programming language with strong static typing, to offer more opportunities for reasoning, abstraction, and static code generation. More specifically, we improve upon two features: *pattern matching* and *type classes*.

Pattern matching denotes the act of performing case-analysis on the shape of a value, thus enabling a program to behave differently, depending on input information. With the advent of extensions such as Generalized Algebraic Data Types (GADTs), pattern guards, and view patterns, the task of reasoning about pattern matching has become much more complex. Though existing approaches can deal with some of these features, no existing algorithm can accurately reason about pattern matching in the presence of all of them, thus hindering the ability of the compiler to guide the development process. The first part of this thesis presents a short, easy-to-understand, and modular algorithm which can reason about lazy pattern matching with GADTs and guards. We have implemented our algorithm in the Glasgow Haskell Compiler.

The second part of this thesis extends upon the design of type classes, one of the most distinctive features of Haskell for function overloading and type-level programming. We develop three independent extensions that lift the expressive power of type classes from simple Horn clauses to a significant fragment of first-order logic, thus offering more possibilities for expressive type-level programming and automated code generation.

The first feature, *Quantified Class Constraints*, lifts the language of constraints from simple Horn clauses to Harrop formulae. It significantly increases the modelling power of type classes, while at the same time it enables a terminating type class resolution for a larger class of applications.

The second feature, *Functional Dependencies*, extends type classes with implicit type-level functions. Though functional dependencies have been implemented in Haskell compilers for almost two decades, several aspects of their semantics have been ill-understood. Thus, existing implementations of functional dependencies significantly deviate from their specification. We present a novel formalization of functional dependencies that addresses all such problems, and give the first type inference algorithm for functional dependencies that successfully elaborates the feature into a statically-typed intermediate language.

The third feature, *Bidirectional Instances*, allows for the interpretation of class instances bidirectionally, thus indirectly adding the biconditional connective in the language of constraints. This extension significantly improves the interaction of GADTs and type classes, since it allows functions with qualified types to perform structural induction over indexed types. Moreover, under this interpretation class-based extensions such as functional dependencies and associated types become much more expressive.

In summary, this thesis extends upon existing and develops new type-level features, promoting the usage of rich types that can capture and statically enforce program properties.

---

https://doi.org/10.1017/S0956796818000175 Published online by Cambridge University Press

Tools for Discovery, Refinement and Generalisation of Functional Properties by Enumerative Testing

RUDY MATELA BRAQUEHAIS
University of York, UK

Date: October 2017; Advisor: Colin Runciman
URL: https://tinyurl.com/yan9pjaj

This thesis presents techniques for discovery, refinement and generalization of properties about functional programs. These techniques work by reasoning from test results: their results are surprisingly accurate in practice, despite an inherent uncertainty in principle. These techniques are validated by corresponding implementations in Haskell and for Haskell programs: Speculate, FitSpec and Extrapolate. Speculate discovers properties given a collection of black-box function signatures. Properties discovered by Speculate include inequalities and conditional equations. These properties can contribute to program understanding, documentation and regression testing. FitSpec guides refinements of properties based on results of black-box mutation testing. These refinements include completion and minimization of property sets. Extrapolate generalizes counterexamples of test properties. Generalized counterexamples include repeated variables and side-conditions and can inform the programmer what characterizes failures. Several example applications demonstrate the effectiveness of Speculate, FitSpec and Extrapolate.

Mechanizing Confluence: Automated and Certified Analysis of First- and Higher-Order Rewrite Systems

JULIAN NAGELE
University of Innsbruck, Austria

Date: March 2018; Advisor: Aart Middeldorp
URL: https://tinyurl.com/yaft3tgd

This thesis is devoted to the mechanized confluence analysis of rewrite systems. Rewrite systems consist of directed equations and computation is performed by successively replacing instances of left-hand sides of equations by the corresponding instance of the right-hand side. Confluence is a fundamental property of rewrite systems, which ensures that different computation paths produce the same result. Since rewriting is Turing-complete, confluence is undecidable in general. Nevertheless, techniques have been developed that can be used to determine confluence for many rewrite systems and several automatic confluence provers are under active development.

Our goal is to improve three aspects of automatic confluence analysis, namely (a) reliability, (b) power, and (c) versatility. The importance of these aspects is witnessed by the annual international confluence competition, where the leading automated tools analyze confluence of rewrite systems. To improve the *reliability* of automatic confluence analysis, we formalize confluence criteria for rewriting in the proof assistant Isabelle/HOL, resulting in a verified, executable checker for confluence proofs. To enhance the *power* of confluence tools, we present a remarkably simple technique, based on the addition and removal of redundant equations, that strengthens existing techniques. To make automatic confluence analysis more *versatile*, we develop a higher-order confluence tool, making automatic confluence analysis applicable to systems with functional and bound variables.

Denotational Semantics in Synthetic Guarded Domain Theory

MARCO PAVIOTTI
IT University of Copenhagen, Denmark

Date: October 2016; Advisor: Rasmus Ejlers Møgelberg and Jesper Bengtson
URL: https://tinyurl.com/y76bpwfj

In functional programming, features such as recursive terms and types and general references are central and these have traditionally been modelled using domain theory. Unfortunately, domain theory does not scale to languages with features such as polymorphism and general references, and this has prevented it from being more widely used.

Step-indexing is a more general technique that has been used to break the circularity of recursive definitions. The idea is to tweak the definition by adding a well-founded structure that gives a handle for recursion. Guarded dependent Type Theory (GdTT) is a type theory which implements step-indexing via a unary modality used to guard recursive definitions. Every circular definition is well-defined as long as the recursive variable is *guarded*.

In this work, we show that GdTT is a natural setting to give denotational semantics of typed functional programming languages with recursion and recursive types. We formulate operational semantics and denotational semantics and prove computational adequacy entirely *inside the type theory*. We do this by encoding *recursion as guarded recursion* and *recursive types as guarded recursive types*. Working in GdTT also allows us to use guarded recursion as a proof technique, and this is used in the proofs of computational adequacy.

Finally, this work builds the foundations for doing denotational semantics of languages with much more challenging features, e.g. general references, for which denotational techniques were previously beyond reach.

Extensible and Robust Functional Reactive Programming

IVÁN PÉREZ DOMÍNGUEZ
University of Nottingham, UK

Date: Feburary 2018; Advisor: Henrik Nilsson and Graham Hutton
URL: https://tinyurl.com/yawzha3p

Programming GUI and multimedia in functional languages has been a long-term challenge, and no solution convinces the community at large. Purely functional GUI and multimedia toolkits enable abstract thinking, but have enormous maintenance costs.

General solutions like Functional Reactive Programming present a number of limitations. FRP has traditionally resisted efficient implementation, and existing libraries sacrifice determinism and abstraction in the name of performance. FRP also enforces structural constraints that facilitate reasoning, but at the cost of modularity and separation of concerns.

This work addresses those limitations with the introduction of Monadic Stream Functions, an extension to FRP parameterised over a monad. I demonstrate that, in spite of being simpler than other FRP proposals, Monadic Stream Functions subsume and exceed other FRP implementations.

Unlike other proposals, Monadic Stream Functions maintain purity at the type level, which is crucial for testing and debugging. I demonstrate this advantage by introducing FRP testing facilities based on temporal logics, together with debugging tools specific for FRP.

I present two uses cases for Monadic Stream Functions: First, I show how the new constructs improved the design of game features and non-trivial games. Second, I present Reactive Values and Relations, an abstraction for model-view coordination in GUI programs based on a relational language, built on top of Monadic Stream Functions. Comprehensive examples are used to illustrate the benefits of this proposal in terms of clarity, modularity, feature coverage, and its low maintenance costs. The testing facilities mentioned before are used to encode and statically check desired interaction properties.

Tierless Web Programming in ML

GABRIEL RADANNE
Paris Diderot University, France

Date: November 2017; Advisor: Jérôme Vouillon and Roberto Di Cosmo
URL: https://tinyurl.com/yc8q6y3r

Eliom is a dialect of OCaml for Web programming in which server and client pieces of code can be mixed in the same file using syntactic annotations. This allows to build a whole application as a single distributed program, in which it is possible to define in a composable way reusable widgets with both server and client behaviors. Eliom is type-safe, as it ensures that communications are well-behaved through novel language constructs that match the specificity of Web programming. Eliom is also efficient, it provides static slicing which separates client and server parts at compile time and avoids back-and-forth communications between the client and the server. Finally, Eliom supports modularity and encapsulation thanks to an extension of the OCaml module system featuring tierless annotations that specify whether some definitions should be on the server, on the client, or both. This thesis presents the design, the formalization and the implementation of the Eliom language.

Type Error Customization for Embedded Domain-Specific Languages

ALEJANDRO SERRANO MENA
Utrecht University, The Netherlands

Date: April 2018; Advisor: Jurriaan Hage and Johan Jeuring
URL: https://tinyurl.com/y8rnvgtd

Domain-specific languages (DSLs) are a widely used technique in the programming world, since they make communication between experts and developers more fluid. Some well-known examples are SQL for databases and HTML for web page description. There are two different approaches to developing DSLs: external – a new compiler is created from scratch – and internal or embedded – the DSL is a library inside a general-purpose language. We focus on the latter, in particular in DSLs which are embedded in strongly-typed functional languages such as Haskell.

Unfortunately, the use of an embedded DSL is not completely transparent, as it ought to be. Since from the point of view of the compiler the DSL is merely a library, *type error messages* are not phrased in domain terms. Not only is the abstraction thereby broken, but also the internals of the library become exposed. The consequence is that programmers have to learn to decipher the error messages to make use of the DSL in a productive way.

This problem is not new to the community. In fact, Jeremy Wazny in his thesis *Type Inference and Type Error Diagnosis for Hindley/Milner with Extensions* and Bastiaan Heeren in his thesis *Top Quality Type Error Messages* focus on this same problem. This thesis extends their work in three different directions: abstraction, context-dependence, and support for advanced type systems.

The use of error customization requires creators of DSLs to write the error messages to show in each situation. Very often, the error messages for a given library are quite similar. None of the previous work tackled the problem of possible duplication, and *abstraction* over common error patterns. Our solution is two-fold: first, the introduction of more powerful matching mechanisms for errors, such as tree regular expressions and functional patterns, whenever the language designer is able to change the general-purpose language; second, the use of type-level features in Haskell to construct type error messages.

Another common problem in this area is to give different error messages to the same piece of code *depending on the context* in which it appears. For example, Haskell has an "fmap" function which works on lists of values, but also optional values and others. This is usually an advantage, since the programmer only has to learn one function for many different scenarios. But this complicates error reporting. We propose a two-phase type checking process in which the error messages can be refined by the types of the elements involved.

The final research question is whether our aproach can be translated to other languages. To fulfill this goal we need to develop a *shared framework* to describe type systems. The most common approach, Constraint Handling Rules, falls

short to describe modern features such as GADTs. Our first step thus is to extend Constraint Handling Rules with type variables and universal quantification, by reusing techniques from higher-order logic programming. To show that this framework is useful, we describe impredicative types – an advanced feature in Haskell – using our extension to CHRs. In order to prove the confluence and termination of this extension, we introduce the notion of *guarded impredicativity*.

Ultimately, the customization of error messages touches a critical part of the compiler: the type checker. We need to ensure that soundness of the type system is preserved throughout the whole process. We describe a lightweight technique to ensure the soundness of any language extension – of which type error customization is an example – by embedding the languages into Haskell.

In addition to the theoretical work, two prototypes have been build. A milder form of our approach has been integrated into the well-known GHC compiler, enabling us to evaluate our approach with real-world libraries.

https://doi.org/10.1017/S0956796818000175 Published online by Cambridge University Press

Scalable Designs for Abstract Interpretation of Concurrent Programs: Application to Actors and Shared-Memory Multi-Threading

QUENTIN STIÉVENART
Vrije Universiteit Brussel, Belgium

Date: May 2018; Advisor: Coen De Roover and Wolfgang De Meuter
URL: https://tinyurl.com/y8v4febx

Concurrent programs are difficult for developers to reason about. They consist of concurrent processes, of which the execution can be interleaved in an exponential number of ways. Contemporary concurrent programs moreover feature dynamic process creation and termination, exacerbating the need for tool support.

Static program analysis can be a powerful enabler of such tool support, but current static analysis designs for concurrent programs are limited with respect to one or more of the following desirable properties: automation, soundness, scalability, precision, and support for dynamic process creation. Moreover, existing analyses are designed with a single concurrency model in mind, and uniform design methods applicable to multiple concurrency models are lacking.

We study the applicability of a recent design method for static program analyses — abstracting abstract machines (AAM) — to concurrent programming languages. Applying this method results in analyses featuring all desirable properties except scalability. We present MacroConc and ModConc, two AAM-inspired uniform design methods that, when applied to the operational semantics of a concurrent programming language, result in static analyses featuring all the desired properties.

The first design method, MacroConc, introduces Agha's 1997 notion of macro-stepping into AAM analyses for concurrent programs. Refining the default all-interleavings semantics for concurrent processes with macro-stepping reduces the number of interleavings the analysis has to explore. The resulting analyses remain exponential in their worst-case complexity, but mitigate the scalability issues of existing analyses without compromising their precision. The second design method, ModConc, introduces Cousot and Cousot's 2002 notion of modularity into AAM analyses for concurrent programs. Analyses resulting from this design method consider each process of a concurrent program in isolation to infer potential interferences with other and newly created processes, which will have to be reconsidered until a fixed point is reached. Process interleavings are not explicitly modeled by the analysis but still accounted for. This analysis design trades off precision to yield process-modular analyses that scale linearly with the number of processes created in the program under analysis.

To demonstrate generality, we apply each design method to two prominent concurrent programming models: concurrent actors and shared-memory multi-threading. We prove the soundness and termination properties for each of the resulting analyses formally, and evaluate their running times, scalability and precision empirically on a set of 56 concurrent benchmark programs. Analyses resulting from

the application of MacroConc achieve high precision, yet exhibit a reduction in running time of up to four orders of magnitude, compared to analyses resulting from a naive application of AAM. Analyses resulting from the application of ModConc exhibit a more consistent reduction, compared to both analyses resulting from the application of AAM and of MacroConc, but this at the cost of lower precision.

The complementary design methods presented in this dissertation enable one to select an analysis design fitting their needs in terms of scalability and precision, enabling future tool support for contemporary concurrent programs.

---

https://doi.org/10.1017/S0956796818000175 Published online by Cambridge University Press