JFP **25**, e21, 37 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S0956796815000283

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 25 abstracts from 2014/15 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

The Nax language: Unifying functional programming and logical reasoning in a language based on Mendler-style recursion schemes and term-indexed types

KI YUNG AHN
Portland State University, USA

Date: December 2014; Advisor: Tim Sheard
URL: `http://archives.pdx.edu/ds/psu/13198`

Two major applications of typed lambda calculi in computer science are functional programming languages and mechanized logical reasoning systems (a.k.a., proof assistants, interactive theorem prover). This dissertation particularly focuses on higher-order & polymorphically typed functional languages and dependently typed reasoning systems for higher-order logic based on the Curry–Howard correspondence. According to the well-known slogan of the Curry–Howard correspondence, *"propositions as types and proofs as programs"*, it should be possible in principle to design a unified language based on a typed lambda calculus for both logical reasoning and programming. However, the different requirements of programming languages and reasoning systems make it difficult to design such a unified language that provides features to meet the needs of both. Programming languages usually extend lambda calculi with programming-friendly features (e.g., recursive datatypes, general recursion) for supporting the flexibility to model various computations, while sacrificing logical consistency. Logical reasoning systems usually extend lambda calculi with logic-friendly features (e.g., induction principles, dependent types) for paradox-free reasoning over fine-grained properties, while being more restrictive in modeling computations.

In this dissertation, we design and implement a language called Nax that embraces benefits of both. The design of Nax is aimed at the sweet spot for unifying programming and reasoning with the following four desirable characteristics:

A convenient programming style supported by the major constructs of modern functional programming languages: parametric polymorphism, recursive datatypes, recursive functions, and type inference,

An expressive logic that can specify fine-grained program properties using types, and terms that witness the proofs of these properties (Curry–Howard correspondence),

A small theory based on a minimal foundational calculus that is expressive enough to support programming features, expressive enough to embed propositions and proofs about programs, and logically consistent to avoid paradoxical proofs in the logic, and

A simple implementation that keeps the trusted base small. This is possible because we designed Nax with type-based termination checking, which does not need to implement a termination checker separately from the type system.

The key features of Nax listed below supports the four characteristics discussed above:

All recursive datatypes including both positive and negative recursive occurrences are supported in Nax, allowing the same flexibility of defining recursive datatypes as in functional languages. Most dependently typed reasoning systems based on the Curry–Howard correspondence only support strictly-positive datatypes, which can be interpreted as sets, whereas functional languages support support recursive types of arbitrary recursive occurrences. For instance, the recursive datatype definition for the untyped higher-order abstract syntax **data** $Exp = App\ Exp\ Exp \mid Abs\ (Exp \rightarrow Exp)$ in Haskell is not accepted as well-formed inductive datatype definitions in Coq or Agda in order to avoid paradoxical proofs. Nax supports such recursive types and yet maintains logical consistency.

A number of Mendler-style recursion schemes in Nax can express various kinds of recursive computations and also guarantee termination. Logical reasoning systems establish the Curry–Howard correspondence assuming normalization to ensure logical consistency. One way to ensure termination of recursive computation is to restrict the class of recursive datatypes where one can assign obvious set-theoretic size measures over the values of the datatypes, interpreting types as sets—this approach is taken by most of the contemporary reasoning systems. On the contrary, Nax accepts all recursive datatypes but ensures termination by relying on more carefully tailored recursion schemes that ensure termination in a type based way—one such Mendler-style recursion scheme was first discovered by Nax Mendler, hence, named as the Mendler style. Mendler-style recursion schemes can ensure termination over recursive datatypes with negative occurrences. In addition, they generalize naturally to non-regular datatypes including term-indexed datatypes.

Term-indexed types in Nax supports specifications of fine-grained properties. In addition to the datatypes that are indexed by types (e.g., type-indexed expression datatype used for writing type-preserving evaluator with a static guarantee of type preservation), Nax supports datatypes that are indexed by terms (e.g., length-indexed list datatype). To support term-indices in a logically consistent manner, we formulated new typed lambda calculi that conservatively extend polymorphic lambda calculi, which are known to be logically consistent, with erasable term indices.

A conservative extension of the Hindley–Milner type inference is supported in Nax for programmer convenience. All programs that are type-inferable with the Hindley–Milner type inference can also be type-inferred in Nax. Programs involving indexed datatypes need type annotations at the elimination (i.e., pattern matching in case expressions and Mendler-style recursion schemes) over the values of indexed datatypes.

The theoretical contributions of this dissertation include theories for Mendler-style recursion schemes and term-indexed types, which we developed to establish strong normalization and logical consistency of Nax. The design and implementation of Nax demonstrate that it is possible to support both programming and reasoning in a unified language with a simple theory and implementation. Candidates that can practically benefit from applying our approach are functional languages like

Haskell, which have pragmatic support for *lightweight program invariants* using the language type system but lacks the guarantee of logical consistency. By applying our approach in Nax, high assurance of logical consistency, comparative to the level of assurance provided in the mechanized reasoning systems, can be provided with a relatively low cost, that is, relatively mild extensions to the language type system without having to introduce full-dependent types found in proof assistants.

We hope to inspire practical programmer friendly mechanized reasoning systems, which is also a programming language at the same time, by further work in line of this thesis research. Here are some of the research problems for further work:

1. even more expressive term-indexed calculi (e.g., kind polymorphism, type representations for datatype generic programming) while maintaining their logical consistency,
2. establishing a unified termination argument when several different Mendler-style recursion schemes are used altogether—some recursion schemes have compatible semantic embeddings into a typed lambda calculus but for other combinations of them yet needs further clarification, although each individual recursion scheme has its termination property established, and, in addition,
3. further list of newly formulated Mendler-style recursion schemes whose termination properties are yet to be studied and to be compared with other approaches to type-based termination.

Digital circuits in CλaSH: Functional specifications and type-directed synthesis

CHRISTIAAN BAAIJ
University of Twente, The Netherlands

Date: *January 2015;* *Advisor:* *Gerard Smit and Jan Kuper*
URL: `http://dx.doi.org/10.3990/1.9789036538039`

Over the last three decades, the number of transistors used in microchips has increased by three orders of magnitude, from millions to billions. The productivity of the designers, however, lags behind. Designing a chip that uses ever more transistors is complex, but doable, and is achieved by massive replication of functionality. Managing to implement complex algorithms, while keeping non-functional properties, such as area and gate propagation latency, within desired bounds, and thoroughly verifying the design against its specification, are the main difficulties in circuit design.

It is difficult to measure design productivity *quantitatively*; transistors per hour would not be a good measure, as high transistor counts can be achieved by replication. As a motivation for our work, we make a *qualitative* analysis of the tools available to circuit designers. Furthermore, we show how these tools manage the complexity, and hence improve productivity. Here, we see that progress has been slow, and that the same techniques have been used for over 20 years. Industry standard languages, such as VHDL and (System)Verilog, do provide means for abstractions, but they are distributed over separate language constructs and have ad hoc limitations. What is desired is a single abstraction mechanism that can capture most, if not all, common design patterns. Once we can abstract our common patterns, we can reason about them with rigor. Rigorous analysis enables us to develop correct-by-construction transformations that capture trade-offs in the non-functional properties. These correct-by-construction transformations give us a straightforward path to reaching the desired bounds on non-functional properties, while significantly reducing the verification burden.

We claim that functional languages can be used to raise the abstraction level in circuit design. Especially higher-order functional languages, where functions are first-class and can be manipulated by other functions, offer a single abstraction mechanism that can capture many design patterns. An additional property of functional languages that make them a good candidate for circuit design is purity, which means that functions have no side-effects. When functions are pure, we can reason about their composition and decomposition locally, thus enabling us to reason formally about transformations on these functions. Without side-effects, synthesis can derive highly parallel circuits from a functional description because it only has to respect the direct data dependencies.

In existing work, the functional language *Haskell* has been used as a host for *embedded* hardware description languages. An embedded language is actually a set

of data types and expressions described within the host language. These data types and expressions then act like the keywords of the embedded language. Functions in the host language are subsequently used to model functions in the embedded language. Although many features of the host language can be used to model equivalent behavior in the embedded language, this is not true for all features. One of the most important features of the host language that cannot directly be used in the embedded language, are features that model choice, such as pattern matching.

This thesis explores the idea of using the functional language Haskell *directly* as a hardware specification language, and move beyond the limitations of embedded languages. Additionally, where applicable, we can use normal functions from existing Haskell libraries to model the behavior of our circuits.

There are multiple ways to interpret a function as a circuit description. This thesis makes the choice of interpreting a function definition as a *structural* composition of components. This means that every function application is interpreted as the component instantiation of the respective sub-circuit. Combinational circuits are then described as functions manipulating algebraic data types. Synchronous sequential circuits are described as functions manipulating infinite streams of values. In order to reduce the cognitive burden, and to guarantee synthesizable results, streams cannot be manipulated directly by the designer. Instead, our system offers a limited set of combinators that can safely manipulate streams, including combinators that map combinational functions over streams. Additionally, the system offers streams that are explicitly synchronized to a particular clock and thus enable the design of multi-clock circuits. Proper synchronization between clock domains is checked by the type system.

This thesis describes the inner workings of our CλaSH *compiler*, which translates the aforementioned circuit descriptions written in Haskell to low-level descriptions in VHDL. Because the compiler uses Haskell directly as a specification language, synthesis of the description is based on (classic) static analysis. The challenge then becomes the reduction of the higher-level abstractions in the descriptions to a form where synthesis is feasible. This thesis describes a term rewrite system (with bound variables) to achieve this reduction. We prove that this term rewrite system *always* reduces a polymorphic, higher-order circuit description to a synthesizable variant. The only restriction is that the root of the function hierarchy is not polymorphic nor higher order. There are, however, no restrictions on the use of polymorphism and higher-order functionality in the rest of the function hierarchy.

Even when descriptions use high-level abstractions, the CλaSH compiler can synthesize efficient circuits. Case studies show that circuits designed in Haskell, and synthesized with the CλaSH compiler, are on par with hand-written VHDL, in both area and gate propagation delay. Even in the presence of contemporary Haskell idioms and abstractions to write imperative code (for a control-oriented circuit), does the CλaSH compiler create results with decent non-functional properties. To emphasize that our approach enables correct-by-construction descriptions, we demonstrate abstractions that allow us to automatically compose components that use back-pressure as their synchronization method. Additionally, we show how cycle delays can be encoded in the type-signatures of components, allowing us to catch any synchronization error at compile-time.

This thesis thus shows the merits of using a modern functional language for circuit design. The advanced type system and higher-order functions allow us to design circuits that have the desired property of being correct-by-construction. Finally, our synthesis approach enables us to derive efficient circuits from descriptions that use high-level abstractions.

The productivity of polymorphic stream equations and the composition of circular traversals

FLORENT BALESTRIERI
University of Nottingham, UK

Date: October 2015; Advisor: Venanzio Capretta
URL: `http://eprints.nottingham.ac.uk/29745`

This thesis explores two aspects of lazy functional programs. The first part is a theoretical study of productivity for very restricted stream programs. The second part is concerned with the elaboration of a recursive pattern for defining circular traversals modularly.

Productivity is in general undecidable. By restricting ourselves to mutually recursive polymorphic stream equations having only three basic operations, namely `head`, `tail`, and `cons`, we aim to prove interesting properties about productivity. Still undecidable for this restricted class of programs, productivity of polymorphic stream functions is equivalent to the totality of their indexing functions, which characterise their behavior in terms of operations on indices. We prove that our equations generate all possible polymorphic stream functions, and therefore their indexing functions are all the computable functions, whose totality problem is indeed undecidable. We then further restrict our language by reducing the numbers of equations and parameters, but despite those constraints the equations retain their expressiveness. In the end, we establish that even two non-mutually recursive equations on unary stream functions are undecidable with complexity Π^0_2. However, the productivity of a single unary equation is decidable.

Circular traversals have been used in the eighties as an optimization to combine multiple traversals in a single traversal. In particular, they provide more opportunities for applying deforestation techniques since it is the case that an intermediate data structure can only be eliminated if it is consumed only once. Another use of circular programs is in the implementation of attribute grammars in lazy functional languages. There is a systematic transformation to define a circular traversal equivalent to multiple traversals. Programming with this technique is not modular since the individual traversals are merged together. Some tools exist to transform programs automatically and attribute grammars have been suggested as a way to describe the circular traversals modularly. Going to the root of the problem, we identify a recursive pattern that allows us to define circular programs modularly in a functional style. We give two successive implementations, the first one is based on algebras and has limited scope: not all circular traversals can be defined this way. We show that the recursive scheme underlying attribute grammars computation rules is essential to combine circular programs. We implement a generic recursive operation on a novel attribute grammar abstraction, using containers as a parametric generic representation of recursive datatypes. The abstraction

makes attribute grammars first-class objects. Such a strongly typed implementation is novel and makes it possible to implement a high-level embedded language for defining attribute grammars, with many interesting new features promoting modularity.

On the incremental evaluation of higher-order attribute grammars

JEROEN BRANSEN
Utrecht University, the Netherlands

Date: June 2015; Advisor: Doaitse Swierstra and Atze Dijkstra
URL: `http://dspace.library.uu.nl/handle/1874/313166`

Compilers, amongst other programs, often work with data that (slowly) changes over time. When the changes between subsequent runs of the compiler are small, one would hope the compiler to incrementally update its results, resulting in much lower running time. However, the manual construction of an incremental compiler is hard and error prone and therefore usually not an option. Attribute grammars provide an attractive way of constructing compilers, as they are compositional in nature and allow for aspect-oriented programming. This thesis describes the automatic generation of incremental attribute grammar evaluators, with the purpose of (semi-)automatically generating an incremental compiler from the regular attribute grammar definition. In particular, this approach supports incremental evaluation of higher order attributes, a well-known extension to the classical attribute grammars that is used in many ways in compiler construction, for example to model different compiler phases.

Semantic methods for functional hybrid modeling

JOHN CAPPER
University of Nottingham, UK

Date: December 2014; Advisor: Henrik Nilsson
URL: `http://eprints.nottingham.ac.uk/27759/`

Equation-based modeling languages have become a vital tool in many areas of science and engineering. Functional Hybrid Modeling (FHM) is an approach to equation-based modeling that allows the behavior of a physical system to be expressed as a *modular hierarchy* of undirected equations. FHM supports a variety of advanced language features—such as higher-order models and variable system structure—that sets it apart from the majority of other modeling languages. However, the inception of these new features has not been accompanied by the semantic tools required to effectively use and understand them. Specifically, there is a lack of static safety assurances for dynamic models and the semantics of the aforementioned language features are poorly understood.

Static safety guarantees are highly desirable as they allow problems that may cause an equation system to become unsolvable to be detected early, during compilation. As a result, the use of static analysis techniques to enforce structural invariants (e.g., that there are the same number of equations as unknowns) is now in use in main-stream equation-based languages like Modelica. Unfortunately, the techniques employed by these languages are somewhat limited, both in their capacity to deal with advanced language features and also by the spectrum of invariants they are able to enforce.

Formalizing the semantics of equation-based languages is also important. Semantics allow us to better understand what a program is doing during execution, and to *prove* that this behavior meets with our expectation. They also allow different implementations of a language to agree with one another, and can be used to demonstrate the correctness of a compiler or interpreter. However, current attempts to formalize such semantics typically fall short of describing advanced features, are not compositional, and/or fail to show correctness.

This thesis provides two major contributions to equation-based languages. First, we develop a refined type system for FHM capable of capturing a larger number of structural anomalies than is currently possible with existing methods. Second, we construct a compositional semantics for the discrete aspects of FHM, and prove a number of key correctness properties.

Variational typing and its applications

SHENG CHEN
Oregon State University, USA

Date: July 2015; Advisor: Martin Erwig
URL: `https://ir.library.oregonstate.edu/xmlui/handle/1957/56350`

The study of variational typing originated from the problem of type inference for variational programs, which encode numerous different but related plain programs. In this dissertation, I present a sound and complete type inference algorithm for inferring types of all plain programs encoded in variational programs. The proposed algorithm runs exponentially faster than the strategy of generating all plain programs and applying type inference to them separately. I also present an error-tolerant version of variational type inference to deliver better feedback in the presence of ill-typed plain programs. All presented algorithms require various kinds of variational unification. I prove that all these problems are decidable and unitary, and I develop sound and complete unification algorithms. The idea of variational typing has many applications. As one example, I present how variational typing can be employed to improve the diagnosis of type errors in functional programs, a problem that has been extensively studied.

HERMIT: Mechanized reasoning during compilation in the Glasgow Haskell compiler

ANDREW FARMER
University of Kansas, USA

Date: April 2015; Advisor: Andy Gill
URL: `http://andrewfarmer.name/papers/dissertation.pdf`

It is difficult to write programs which are both correct and fast. A promising approach, functional programming, is based on the idea of using pure, mathematical functions to construct programs. With effort, it is possible to establish a connection between a specification written in a functional language, which has been proven correct, and a fast implementation, via program transformation.

When practiced in the functional programming community, this style of reasoning is still typically performed by hand, by either modifying the source code or using pen-and-paper. Unfortunately, performing such semi-formal reasoning by directly modifying the source code often obfuscates the program, and pen-and-paper reasoning becomes outdated as the program changes over time. Even so, this semi-formal reasoning prevails because formal reasoning is time-consuming, and requires considerable expertise. Formal reasoning tools often only work for a subset of the target language, or require programs to be implemented in a custom language for reasoning.

This dissertation investigates a solution, called HERMIT, which mechanizes reasoning during compilation. HERMIT can be used to prove properties about programs written in the Haskell functional programming language, or transform them to improve their performance. Reasoning in HERMIT proceeds in a style familiar to practitioners of pen-and-paper reasoning, and mechanization allows these techniques to be applied to real-world programs with greater confidence. HERMIT can also re-check recorded reasoning steps on subsequent compilations, enforcing a connection with the program as the program is developed.

HERMIT is the first system capable of directly reasoning about the full Haskell language. The design and implementation of HERMIT, motivated both by typical reasoning tasks and HERMITs place in the Haskell ecosystem, is presented in detail. Three case studies investigate HERMITs capability to reason in practice. These case studies demonstrate that semi-formal reasoning with HERMIT lowers the barrier to writing programs which are both correct and fast.

Application of property-based automated testing techniques to different levels of software testing

MIGUEL ÁNGEL FRANCISCO FERNÁNDEZ
Universidade da Coruña, Spain

Date: July 2015; Advisor: Laura M. Castro
URL: `http://hdl.handle.net/2183/14814`

Testing is one of the key activities in the software development process. In particular, testing activities help to detect defects that otherwise would go unnoticed until the software is deployed. However, unlike other stages in the software development life cycle, such as analysis, design, or implementation, for which there are well defined and widely accepted methodologies and techniques, as well as tools that allow to carry out these tasks, there is a lack of good and complete methodologies, techniques and tools that can be used to conduct software testing in an efficient and effective way.

Hence, companies tend to minimize this stage, underestimating its benefits, due to its intrinsic complexity and the considerable amount of time and resources it usually requires to be performed properly. Therefore, testing activities are often omitted or performed without the necessary rigor. The poor quality of efforts and low quantity of achievements during testing usually has severe consequences. Normally, maintenance is the longest period in the life cycle of a software product. Therefore, reducing the amount of resources dedicated to this task will clearly improve a company's outcome, and also the user experience. The only way of achieving that is to deploy software products which are as free of defects as possible, i.e., that have been exhaustively tested before being put into a production environment.

This thesis presents a purely functional approach, which uses properties to perform software testing, that attempts to alleviate these problems. To do this, new testing methodologies and techniques, integrated in the software development process, have been designed with the aim of improving the efficiency (spending less time in testing) and effectiveness (finding more defects as soon as possible) of software testing. These methodologies and techniques have been adapted to apply to different levels of software testing. Thus, they can be used to perform unit and component testing, checking that each individual component behaves as expected, integration testing, where the interactions between the components that take part in a software system are checked, and system testing, checking the behavior of a software system as a whole.

In addition, a common test specification language has been used, specifically, the programming language Erlang, a functional programming language designed by Ericsson to support distributed, fault-tolerant, and real-time applications. The main reason for using Erlang as a test specification language is its functional nature and its declarative syntax, which makes it a good specification language for writing properties as well as readable and concise models. Moreover, all the new developed

testing methodologies and techniques, which can be used to test software systems using a property-based testing approach using Erlang as a test specification language, have been designed to test software systems, regardless of the structure of the specific system under test or the programming language in which that software system to test is implemented.

Finally, the use of all these new testing methodologies and techniques has been illustrated through a real-world case study, in particular, the VoDKATV system. This software system, currently installed in several hotels and telecommunication environments around the world, provides access to multimedia services (TV channels, video on demand, applications, games, etc.) through different types of devices, such as televisions, computers, tablets, or smart phones. Regarding its architecture, the VoDKATV system is composed of multiple integrated components implemented with different technologies (Java, Erlang, C, etc.). The complexity of this software system has allowed to illustrate how to put into practice each of the new testing methodologies and techniques with a real, complex, and industrial case study.

Distributing abstract machines

OLLE FREDRIKSSON
University of Birmingham, UK

Date: April 2015; Advisor: Dan Razvan Ghica
URL: `http://etheses.bham.ac.uk/6196/`

Today's distributed programs are often written using either explicit message passing or Remote Procedure Calls that are not natively integrated in the language. It is difficult to establish the correctness of programs written this way compared to programs written for a single computer.

We propose a generalization of Remote Procedure Calls that are natively integrated in a functional programming language meaning e.g., that they have support for higher-order calls across node boundaries. There are already several languages that provide this feature, but there is a lack of details on how they can be compiled correctly and efficiently, which is what this thesis focusses on.

We present four different solutions, given as readily implementable abstract machines. Two of them are based on interaction semantics—the Geometry of Interaction and game semantics—and two of them are moderate extensions of conventional abstract machines—the Krivine machine and the SECD machine. To target general distributed systems, our solutions additionally support higher-order Remote Procedure Calls *without sending actual code*, since this is not generally possible when running on heterogeneous systems in a statically compiled setting.

We prove the correctness of the abstract machines with respect to their single-node execution, and show their viability for use as the basis for compilation by implementing prototype compilers based on them. In the case of the machines based on conventional machines, we additionally show that they enable efficient programs and that single-node performance is not lost when they are used.

Our intention is that these abstract machines can form the foundation for future programming languages that use the idea of higher-order Remote Procedure Calls.

Parsing with regular expressions & extensions to Kleene algebra

NIELS BJØRN BUGGE GRATHWOHL
University of Copenhagen, Denmark

Date: November 2015; Advisor: Fritz Henglein
URL: `http://tinyurl.com/o7ro6u3`

In the first part of this thesis, we investigate methods for regular expression parsing.

We present an $O(mn)$ two-pass algorithm for greedy regular expression parsing in a semi-streaming fashion for expressions of size m and input of size n. Pass one outputs a log of k bits per input symbol, where k is the number of alternatives and Kleene stars in the expression. This log is used in the second pass to produce a full parse tree.

The two-pass algorithm is extended to an $O(2^{m \log m} + mn)$-time optimally streaming parsing algorithm: parts of the parse tree are output as soon as it is semantically possible to do so. To be optimal, the algorithm performs a PSPACE-complete preprocessing step; for a fixed RE the running time is linear in the input size.

Finally, we present and implement a determinization procedure, omitting the preprocessing step, and a surface language, Kleenex, for expressing general string transductions. We have implemented a compiler that translates Kleenex programs into efficient C code. The resulting programs are essentially optimally streaming, run in worst-case linear time in the input size, and show consistent high performance in the 1 Gbps range on various use cases.

In the second part of this thesis, we study two extensions to Kleene algebra.

Chomsky algebra is an algebra with a structure similar to Kleene algebra, but with a generalized mu-operator for recursion instead of the Kleene star. We show that the axioms of idempotent semirings along with continuity of the mu-operator completely axiomatizes the equational theory of the context-free languages.

KAT+B! is an extension to Kleene algebra with tests (KAT) that adds mutable state. We describe a test algebra B! for mutable tests and give a commutative coproduct between KATs. Combining the axioms of B! with those of KAT and some commutativity conditions completely axiomatizes the equational theory of an arbitrary KAT enriched with mutable state.

Truncation levels in homotopy type theory

NICOLAI KRAUS
University of Nottingham, UK

Date: July 2015; Advisor: Thorsten Altenkirch
URL: `http://eprints.nottingham.ac.uk/id/eprint/28986`

Homotopy type theory (HoTT) is a branch of mathematics that combines and benefits from a variety of fields, most importantly homotopy theory, higher-dimensional category theory, and, of course, type theory. We present several original results in HoTT which are related to the truncation level of types, a concept due to Voevodsky.

To begin, we give a few simple criteria for determining whether a type is 0-truncated (a *set*), inspired by a well-known theorem by Hedberg, and these criteria are then generalized to arbitrary n. This naturally leads to a discussion of functions that are weakly constant, i.e., map any two inputs to equal outputs. A weakly constant function does in general not factor through the propositional truncation of its domain, something that one could expect if the function really did not depend on its input. However, the factorization is always possible for weakly constant *endo*functions, which makes it possible to define a propositional notion of anonymous existence. We additionally find a few other non-trivial special cases in which the factorization works. Further, we present a couple of constructions which are only possible with the judgmental computation rule for the truncation. Among these is an invertibility puzzle that seemingly inverts the canonical map from Nat to the truncation of Nat, which is perhaps surprising as the latter type is equivalent to the unit type.

A further result is the construction of strict n-types in Martin-Lof type theory with a hierarchy of univalent universes (and without higher inductive types), and a proof that the universe $U(n)$ is not n-truncated. This solves a hitherto open problem of the 2012/13 special year program on Univalent Foundations at the Institute for Advanced Study (Princeton).

The main result of this thesis is a generalized universal property of the propositional truncation, using a construction of *coherently constant* functions. We show that the type of such coherently constant functions between types A and B, which can be seen as the type of natural transformations between two diagrams over the simplex category without degeneracies (i.e., finite non-empty sets and strictly increasing functions), is equivalent to the type of functions with the truncation of A as domain and B as codomain. In the general case, the definition of natural transformations between such diagrams requires an infinite tower of conditions, which exists if the type theory has *Reedy limits* of diagrams over the ordinal omega. If B is an n-type for some given finite n, (non-trivial) Reedy limits are unnecessary, allowing us to construct functions from the truncation of A to B in HoTT without further assumptions. To obtain these results, we develop some theory on equality

diagrams, especially equality semi-simplicial types. In particular, we show that the semi-simplicial equality type over any type satisfies the Kan condition, which can be seen as the simplicial version of the fundamental result by Lumsdaine, and by van den Berg and Garner, that types are weak omega-groupoids.

Finally, we present some results related to formalizations of infinite structures that seem to be impossible to express internally. To give an example, we show how the simplex category can be implemented so that the categorical laws hold strictly. In the presence of *very dependent types*, we speculate that this makes the Reedy approach for the famous open problem of defining semi-simplicial types work.

Quotient types in type theory

NUO LI
University of Nottingham, UK

Date: July 2015; Advisor: Thorsten Altenkirch
URL: http://eprints.nottingham.ac.uk/28941/

Martin-Lof's intuitionistic type theory (Type Theory) is a formal system that serves not only as a foundation of constructive mathematics but also as a dependently typed programming language. Dependent types are types that depend on values of other types. Type Theory is based on the Curry–Howard isomorphism which relates computer programs with mathematical proofs so that we can do computer-aided formal reasoning and write certified programs in programming languages like Agda, Epigram, etc. Martin Lof proposed two variants of Type Theory which are differentiated by the treatment of equality. In Intensional Type Theory, propositional equality defined by identity types does not imply definitional equality, and type checking is decidable. In Extensional Type Theory, propositional equality is identified with definitional equality which makes type checking undecidable. Because of the good computational properties, Intensional Type Theory is more popular, however it lacks some important extensional concepts such as functional extensionality and quotient types.

This thesis is about quotient types. A quotient type is a new type whose equality is redefined by a given equivalence relation. However, in the usual formulation of Intensional Type Theory, there is no type former to create a quotient. We also lose canonicity if we add quotient types into Intensional Type Theory as axioms. In this thesis, we first investigate the expected syntax of quotient types and explain it with categorical notions. For quotients which can be represented as a setoid as well as defined as a set without a quotient type former, we propose to define an algebraic structure of quotients called definable quotients. It relates the setoid interpretation and the set definition via a normalization function which returns a normal form (canonical choice) for each equivalence class. It can be seen as a simulation of quotient types and it helps theorem proving because we can benefit from both representations. However, this approach cannot be used for all quotients. It seems that we cannot define a normalization function for some quotients in Type Theory, e.g., Cauchy reals and finite multisets. Quotient types are indeed essential for formalization of mathematics and reasoning of programs. Then, we consider some models of Type Theory where types are interpreted as structured objects such as setoids, groupoids, or weak omega-groupoids. In these models, equalities are internalized into types which means that it is possible to redefine equalities. We present an implementation of Altenkirch's setoid model and show that quotient types can be defined within this model. We also describe a new extension of Martin-Lof type theory called Homotopy Type Theory where types are interpreted as weak omega-groupoids. It can be seen as a generalization of the groupoid model which

makes extensional concepts including quotient types available. We also introduce a syntactic encoding of weak omega-groupoids which can be seen as a first step towards building a weak omega-groupoids model in Intensional Type Theory. All of these implementations were performed in the dependently typed programming language Agda which is based on intensional Martin-Lof type theory.

Reduction spaces in non-sequential and infinitary rewriting systems

CARLOS ALBERTO LOMBARDI
Universidad de Buenos Aires, Argentina and Université Paris 7, France

Date: November 2014; Advisor: Delia Nora Kesner,
Alejandro Norberto Ríos and Eduardo Augusto Bonelli
URL: `http://tinyurl.com/qeqly6r`

We study different aspects related to the reduction spaces of diverse rewriting systems. These systems include features which make the study of their reduction spaces a far from trivial task. The main contributions of this thesis are:

1. we define a multistep reduction strategy for the *Pure Pattern Calculus*, a non-sequential higher-order term rewriting system, and we prove that the defined strategy is normalizing;
2. we propose a formalization of the concept of standard reduction for the *Linear Substitution Calculus*, a calculus of explicit substitutions whose reductions are considered modulo an equivalence relation defined on the set of terms, and we obtain a result of *uniqueness* of standard reductions for this formalization; and finally,
3. we characterise the equivalence of reductions for the infinitary, first-order, left-linear term rewriting systems, and we use this characterization to develop an alternative proof of the *compression* result.

We remark that we use *generic models of rewriting systems*: a version of the notion of *Abstract Rewriting Systems* is used for the study of the Pure Pattern Calculus and the Linear Substitution Calculus, while a model based on the concept of *proof terms* is used for the study of infinitary rewriting. We include extensions of both used generic models; these extensions can be considered as additional contributions of this thesis.

Extensible proof engineering in intensional type theory

GREGORY MICHAEL MALECHA
Harvard University, USA

Date: November 2014; Advisor: Greg Morrisett and Adam Chlipala
URL: `http://dash.harvard.edu/handle/1/17467172`

We increasingly rely on large, complex systems in our daily lives—from the computers that park our cars to the medical devices that regulate insulin levels to the servers that store our personal information in the cloud. As these systems grow, they become too complex for a person to understand, yet it is essential that they are correct. Proof assistants are tools that let us specify properties about complex systems and build, maintain, and check proofs of these properties in a rigorous way. Proof assistants achieve this level of rigor for a wide range of properties by requiring detailed certificates (proofs) that can be easily checked.

In this dissertation, I describe a technique for compositionally building extensible automation within a foundational proof assistant for intensional type theory. My technique builds on computational reflection—where properties are checked by verified programs—which effectively bridges the gap between the low-level reasoning that is native to the proof assistant and the interesting, high-level properties of real systems. Building automation within a proof assistant provides a rigorous foundation that makes it possible to compose and extend the automation with other tools (including humans). However, previous approaches require using low-level proofs to compose different automation which limits scalability. My techniques allow for reasoning at a higher level about composing automation, which enables more scalable reflective reasoning. I demonstrate these techniques through a series of case studies centered around tasks in program verification.

Tools for reasoning about effectful declarative programs

STEFAN MEHNER
Rheinische Friedrich-Wilhelms-Universität Bonn, Germany

Date: October 2015; Advisor: Janis Voigtländer
URL: `http://hss.ulb.uni-bonn.de/2015/4178/4178.htm`

In the pure functional language Haskell, nearly all side effects that a function can produce have to be noted in its type. This includes input/output, propagation of a state, and non-determinism. If no side-effects are noted, such a function acts like a mathematical function, i.e., mapping arguments to unique results. In that case, expressions in a program can be reasoned about like mathematical expressions. In addition to this so-called equational reasoning, the type system also enables type based reasoning. One example are free theorems—equations between expressions that are true only due to the types of the expressions involved. Some such statements serve as formal justification for optimization strategies in compilers.

The thesis at hand investigates two generalizations of such methods for programs not free of side-effects, i.e., effectful programs. First, effectful traversals of data structures are being studied. The most important contribution in this part is that a data structure can be lawfully traversed if, and only if, it is isomorphic to a polynomial functor. This result links the widespread interface of traversing to a clear intuition regarding the structure and behavior of the data type. Furthermore, tools are presented facilitating convenient proofs about effectful traversals.

Second, free theorems for the functional-logic language Curry are derived. Due to the close relationship between both languages, Curry can be understood as Haskell with built-in non-determinism, i.e., a built-in side-effect. Equational and type based reasoning can both be adapted to Curry to a certain degree. In particular, short cut fusion—a very fertile runtime optimization—is enabled for Curry.

Semantics-driven design and implementation of high-assurance hardware

ADAM PROCTER
University of Missouri, USA

Date: December 2014; Advisor: William L. Harrison
URL: `https://hdl.handle.net/10355/45775`

Modularity, that is the division of complex systems into less complex and more easily understood parts, is a pervasive concern in computer science, and hardware design is no exception. Existing hardware design languages such as Verilog and VHDL support modular design by enabling hardware designers to decompose designs into *structural* features that may be developed independently and connected together to form more complex devices. In the realm of high assurance for security, however, this sort of modularity is often of limited utility. Security properties are notoriously non-compositional, i.e., subsystems that independently satisfy some security property cannot necessary be relied upon to maintain that property when operating in tandem.

The aim of this research is to establish *semantically modular* techniques for hardware design and implementation, in contrast to the conventional structural notion of modularity. A semantically modular design is constructed by adding "layers" of semantic features, such as state and reactivity, one at a time. From the high assurance aspect, semantic modularity enables different layers of semantic features to be reasoned about independently, greatly simplifying the structure of correctness proofs and improving their reusability. The major contribution of this work is a prototype compiler called ReWire which translates semantically modular hardware specifications to efficient implementations on FPGAs. In this dissertation, I present the design and implementation of the ReWire compiler, along with a number of case studies illustrating both the practicality of the ReWire compiler and the elegance of the semantically modular approach to hardware verification.

On the complexities of polymorphic stream equation systems, isomorphism of finitary inductive types, and higher homotopies in univalent universes

CHRISTIAN SATTLER
University of Nottingham, UK

Date: March 2015; Advisor: Venanzio Capretta
URL: `http://eprints.nottingham.ac.uk/28111/`

This thesis is composed of three separate chapters. Parts of these are based on respective joint work with Florent Balestrieri and Nicolai Kraus.

The first chapter deals with definability and productivity issues of equational systems defining polymorphic stream functions. The central construction is a novel method for encoding arbitrary computable unary polymorphic stream functions in terms of unary non-mutually corecursive polymorphic stream function equation systems. This improves previous completeness and complexity theoretic results, which crucially relied on the availability of so-called zipping (interleaving) non-unary stream functions.

The second chapter deals with syntactic and semantic notions of isomorphism of finitary inductive types and associated decidability issues. We show isomorphism of so-called guarded types decidable in the set and syntactic model, verifying that the answers coincide. The technique relies on finitiary comparison of power series by exhibiting them as roots of polynomials over a field of fractions of polynomials.

In the process, we develop several independent tools, one of them being traversability of regular functors in any bicartesian-closed category. Previous work either relied on properties of regular functors specific to set-like models with sufficient colimits or was restricted to traversals with respect to monads as opposed to applicative functors.

The third chapter deals with homotopy levels of hierarchical univalent universes in homotopy type theory, showing that the n-th universe of n-types has truncation level strictly $n + 1$. The proof uses strategy an inductive argument involving higher loops in appropriately truncated higher universes. Our arguments are formalized in the experimental proof assistant Agda.

Abstractions for software-defined networks

COLE SCHLESINGER
Princeton University, USA

Date: June 2015; Advisor: David Walker
URL: `http://tinyurl.com/ol9a2j4`

In a Software-Defined Network (SDN), a central, computationally powerful controller manages a set of distributed, computationally simple switches. The controller computes a policy describing how each switch should route packets and populates packet-processing tables on each switch with rules to enact the routing policy. As network conditions change, the controller continues to add and remove rules from switches to adjust the policy as needed.

Recently, the SDN landscape has begun to change as several proposals for new, reconfigurable switching architectures, such as RMT and FlexPipe, have emerged. These platforms provide switch programmers with many flexible tables for storing packet-processing rules, and they offer programmers control over the packet fields that each table can analyze and act on. These reconfigurable switch architectures support a richer SDN model in which a switch configuration phase precedes the rule population phase. In the configuration phase, the controller sends the switch a graph describing the layout and capabilities of the packet processing tables it will require during the population phase. Armed with this foreknowledge, the switch can allocate its hardware (or software) resources more efficiently.

This dissertation presents a new, typed language, called Concurrent NetCore, for specifying routing policies and graphs of packet-processing tables. Concurrent NetCore includes features for specifying sequential, conditional, and concurrent control-flow between packet-processing tables. We develop a fine-grained operational model for the language and prove this model coincides with a higher-level denotational model when programs are well-typed. We also prove several additional properties of well-typed programs, including strong normalization and determinism. To illustrate the utility of the language, we develop linguistic models of both the RMT and FlexPipe architectures; give a multi-pass compilation algorithm that translates graphs and routing policies to the RMT model; and evaluate a prototype of the language and compiler on two benchmark applications, a learning switch and a stateful firewall.

Reasoning about functional programs by combining interactive and automatic proofs

ANDRÉS SICARD-RAMÍREZ
Universidad de la República, Uruguay

Date: July 2014; Advisor: Ana Bove and Peter Dybjer
URL: `http://tinyurl.com/qcwkzol`

We propose a new approach to computer-assisted verification of lazy functional programs where functions can be defined by general recursion. We work in first-order theories of functional programs which are obtained by translating Dybjer's programming logic (Dybjer, P. [1985]. Program Verification in a Logical Theory of Constructions. In: Functional Programming Languages and Computer Architecture. Ed. by Jouannaud, J.-P. Vol. 201. Lecture Notes in Computer Science, Springer, pp. 334–349) into a first-order theory, and by extending this programming logic with new (co-)inductive predicates. Rather than building a special purpose system, we formalize our theories in Agda, a proof assistant for dependent type theory which can be used as a generic theorem prover. Agda provides support for interactive reasoning by representing first-order theories using the propositions-as-types principle. Further support is provided by off-the-shelf automatic theorem provers for first-order-logic called by a Haskell program that translates our Agda representations of first-order formulae into the TPTP language understood by the provers. We show some examples where we combine interactive and automatic reasoning, covering both proofs by induction and co-induction. The examples include functions defined by structural recursion, simple general recursion, nested recursion, higher-order recursion, guarded, and unguarded co-recursion.

Data layout types: A type-based approach to automatic data layout transformations for improved SIMD vectorisation

ARTJOMS ŠINKAROVS
Heriot-Watt University, UK

Date: May 2015; Advisor: Sven-Bodo Scholz and Greg Michaelson
URL: `http://tinyurl.com/ocvr2k5`

The Church–Rosser property inherent to purely functional programming languages constitutes a big conceptual advantage when it comes to running programs on parallel systems. It gives rise to a wide range of possibilities for mapping redices to computing cores. This, in turn, gives a lot of freedom for a compiler or interpreter to match the parallelism of any given hardware, which is typically significantly more challenging in the imperative setting.

Despite this conceptual advantage, research over the last decades has shown that it is non-trivial to compete with the performance of hand-optimized codes from classical High-Performance Computing (HPC) domains. Works in the context of functional programming languages like Data Parallel Haskell (DpH) or Single Assignment C (SaC) have demonstrated that this is possible, but it requires advanced compiler technology. One of the key challenges involved when aiming for HPC is the fact that in a functional setting there is no notion of memory. Any data structure is considered to be a mapping from accessors to values; any destructive update of data, at least on a semantic level, is not permissible.

While such a lack of memory poses a challenge in the fist place, it also offers great opportunities when programs are optimized for performance. As the connection between data structures and memory has to be introduced from the underlying execution machinery, this mapping from data to memory can be rather freely chosen. In particular, in the context of HPC, the freedom of the data mapping constitutes a very valuable asset. It is well known from classical HPC research that in many applications non-intuitively chosen data-structures can have a very beneficial impact on performance, specifically, when it comes to code vectorization.

This thesis taps into the optimization opportunities that stem from the conceptual decoupling of data structures and their memory representation in the functional context. We introduce the notion of data layouts to describe the mapping of data structures into memory representations. At the example of SaC we add data layouts transparently to the language semantics. Our main goal here is more efficient vectorization, however, in general, the proposed approach makes it possible to use data layouts as a new degree of freedom in program transformations.

The key idea of this thesis is to treat data layouts as types. Every expression in a program conceptually gets a layout type assigned to it which prescribes the memory mapping of the evaluated expression. By doing so, not only we can check that layout types are sound across the program, but we can also automatically transform original functions into functions with a specific layout-type signature. Finally, and this is the main insight of the thesis, we can run layout-type inference and obtain a set of all the possible semantically preserving program transformations, that are allowed by our layout type system. Note that we have a set of transformed programs, and not just one program, mainly because it is possible to traverse a structure with the modified data layout in its original order.

The contributions of this thesis lie in developing a type system for data layouts that are oriented to improve applicability of SIMD operations. We develop the inference algorithm that makes it possible to reconstruct data layouts according to the type system. We introduce automatic high-level program transformations based on the previously inferred layout types and prove that transformations preserve the semantics of the original programs. We discuss how to chose the best possible program transformation out of the produced set.

We implement the proposed inference, transformation, and generation of the vectorized code in the context of the SaC compiler toolchain. When generating vectorized code, we make sure that we do this in a performance portable fashion. For that we have extended the C language with explicit vector operations which we have implemented in the context of GNU GCC. Finally, we evaluate our approach using a set of benchmarks which is known to be challenging to vectorize and we demonstrate the effectiveness of our vectorization by comparing the runtimes with automatically and manually vectorized C versions. We observe significant performance improvements over all the mainstream C compilers that we have tried.

This thesis demonstrates that layout transformations, which are known to be difficult and error-prone, can be automated by means of data-layout inference. The functional framework has been the key enabling factor for this work. The lack of stateful memory, pure functions, explicit information about concurrent parts of a program, and the ability to run type inference are the basic factors that we have built our solution on. Despite such strong ties with functional environment, we believe that the transition of the proposed technique into imperative languages is possible with a few restrictions and some additional analysis. This means that a wider range of compilers can potentially benefit by adopting the proposed treatment of data layouts.

On the design of finite-state type systems

ALEXANDER IAN SMITH
University of Birmingham, UK

Date: December 2015; Advisor: Dan Razvan Ghica
URL: `http://etheses.bham.ac.uk/6120/`

Practical computers have only finite amounts of memory. However, the programs that run on them are often written in languages that effectively assume (via providing constructs such as general recursion) that infinite memory is available, meaning that an implementation of those programs is necessarily an approximation.

The main focus of this thesis is on the use of contraction: the ability to use a function parameter more than once in the body of that function (or more generally, to mention a free variable more than once in a term). Unrestricted contraction is a common reason for a language to require unbounded amounts of memory to implement.

This thesis looks at a range of type systems, both existing and new, that restrict the use of contraction so that they can be implemented with finite amounts of state, identifying common themes, and explaining and suggesting solutions for common deficiencies. In particular, different restrictions on contraction are seen to correspond to different features of the languages implementation.

How to generate actionable advice about performance problems

VINCENT ST-AMOUR
Northeastern University, USA

Date: May 2015; Advisor: Matthias Felleisen
URL: `http://www.ccs.neu.edu/racket/pubs/dissertation-st-amour.pdf`

Performance engineering is an important activity regardless of application domain, as critical for server software as for mobile applications. This activity, however, demands advanced, specialized skills that require a significant time investment to acquire, and are therefore absent from most programmers' tool-boxes.

My thesis is that tool support can make performance engineering both accessible and time-efficient for non-expert programmers. To support this claim, this dissertation introduces two novel families of performance tools that are designed specifically to provide actionable information to programmers: *optimization coaches* and *feature-specific profilers*. This dissertation presents blueprints for building tools in these families, and provides examples from tools that I have built.

Two topics in rewriting: Combinators for pattern calculi and Curry-Howard for the logic of proofs

GABRIELA STEREN
Universidad de Buenos Aires, Argentina

Date: December, 2014; Advisor: Eduardo A. Bonelli
URL: http://tinyurl.com/nsrcly9

Pattern matching is a basic building block on which functional programming depends, where the computation mechanism is based on finding a correspondence between the argument of a function and an expression called "pattern". It has also found its way into other programming paradigms and has proved convenient for querying data in different formats, such as semi-structured data. In recognition of this, a recent effort is observed in which pattern matching is studied in its purest form, namely by means of pattern calculi. These are lambda calculi with sophisticated forms of pattern matching. The first part of this two part thesis proposes to contribute to this effort by developing a combinatory logic for one such pattern calculus, namely λP. We seek to mimic the computational process of λP where arguments can be matched against arbitrary terms, without the use of variables. Two challenges must be met. On the one hand, dealing with bound variables in patterns. Indeed, an abstract ion is a valid pattern in λP. Here, the standard combinatory logic will provide guidance. The second is computing the counterpart, in the combinatory setting, of the substitution that is obtained in a successful match. This requires devising rules that pull applications apart, so to speak. We propose a combinatory logic that serves this purpose and study its salient properties and extensions including typed presentations and modeling data structures. In the second part, we are concerned with the computational interpretation of a particular modal logic, the Logic of Proofs or LP, via the Curry–Howard isomorphism. LP, introduced by Artemov in 1995, is a refinement of modal logic in which the modality $\Box A$ is revisited as $[\![t]\!]A$, where t is an expression that bears witness to the validity of A. It enjoys arithmetical soundness and completeness, can realize all S4 theorems and is capable of reflecting its own proofs ($\vdash A$ implies $\vdash [\![t]\!]A$, for some t). Our main contribution is a well-behaved Natural Deduction presentation, developed with the aim of unveiling the computational metaphors which arise from the reflective capabilities of LP. This is the first Natural Deduction formulation capable of proving all LP-theorems. For that, we adopt Parigots Classical Natural Deduction and merge it with a hypothetical reasoning which guide the construction of the inference schemes. As an outcome we obtain a Natural Deduction presentation of propositional LP for which a number of key properties are shown to hold. We then extend our analysis to the first-order case, introducing FOHLP, a first-order extension of HLP. Our point of departure is a recent first-order formulation of LP, called FOLP, which enjoys arithmetical soundness and has an exact provability semantics (completeness

is unattainable given that a complete FOLP is not finitely axiomatizable). We provide a Natural Deduction presentation dubbed FOHLP, mappings to and from FOLP, a term assignment (-calculus) and a proof of termination of normalization of derivations.

A transformation-based approach to hardware design using higher-order functions

RINSE WESTER
University of Twente, Netherlands

Date: July 2015; Advisor: Gerard Smit and Jan Kuper
URL: `http://doc.utwente.nl/96278/`

The amount of resources available on reconfigurable logic devices like FPGAs has seen a tremendous growth over the last 30 years. During this period, the amount of programmable resources (CLBs and RAMs) has increased by more than three orders of magnitude.

Programming these reconfigurable architectures has been dominated by the hardware description languages VHDL and Verilog. However, it has become generally accepted that these languages do not provide adequate abstraction mechanisms to deliver the design productivity for designing more and more complex applications. To raise the abstraction level, techniques to translate high-level languages to hardware have been developed based on imperative languages like C.

Parallelism is achieved by parallelization of for-loops. Whether parallelization of loops is possible, is determined using dependency analysis which is a very hard problem. To mitigate this problem, other abstractions are needed to express parallelism. In this thesis, parallelism is expressed using higher-order functions, an abstraction commonly used in functional programming languages.

The main contribution of this thesis is a design methodology based on exploiting regularity of higher-order functions. A mathematical formula, e.g., a DSP algorithm, is first formulated using higher-order functions. Then, transformation rules are applied to these higher-order functions to distribute computations over space and time. Using these transformations, an optimal trade-off can be made between space and time. Finally, hardware is generated using the CLaSH compiler by translating the result of the transformation to VHDL.

In this thesis, we derive transformation rules for several higher-order functions and prove that the transformations are meaning-preserving. After transformation, a mathematically equivalent description is derived in which the computations are distributed over space and time. The designer can control the amount of parallelism using a parameter that is introduced by the transformation. Transformation rules for both one-dimensional higher-order functions and two-dimensional higher- order functions have been derived and applied to several case studies: a dot product, a particle filter and stencil computations.

Effects, asynchrony, and choice in arrowized functional reactive programming

DANIEL WINOGRAD-CORT
Yale University, USA

Date: December 2015; Advisor: Paul Hudak
URL: `http://tinyurl.com/oyt45j7`

Functional reactive programming facilitates programming with time-varying data that can be perceived as streams flowing through time. Thus, one can think of FRP as an inversion of flow control from the structure of the program to the structure of the data itself. In a typical (say, imperative) program, the structure of the program governs how the program will behave over time; as time moves forward, the program sequentially executes its statements, and at any line of code, one can make a clear distinction between code that has already been run (the past) and code that has yet to be run (the future). However, in FRP, the program acts as a signal function, and, as such, we are allowed to assume that the program executes *continuously* on its time-varying inputs—essentially, it behaves as if it is running infinitely fast and infinitely often. We consider this to be the core principle of the design and call it the *fundamental abstraction of FRP*.

This work is specifically rooted in *Arrowized* FRP, where these signal functions remain static as they process the dynamic signals they act upon. However, in practice, it is often valuable to be able to dynamically alter the way that a signal function behaves over time. Typically, this is achieved with "switching" or other monadic features, but this significantly reduces the usefulness of the arrows. We develop an extension to arrows to allow "predictably dynamic" behavior along with a notion of *settability*, which together recover the desired dynamic power. We further demonstrate that optimizations designed specifically for arrowized FRP and which do not apply to monadic FRP, such as those for Causal Commutative Arrows, are applicable to the system. Thus, it can be powerfully optimized.

In its purest form, functional reactive programming permits no side effects (e.g., mutation, state, interaction with the physical world), and as such, all effects must be performed outside of the FRP scope. In practice, this means that FRP programs must route input data streams to where they are internally used and likewise route output streams back out to the edge of the FRP context. I call this the FRP *I/O bottleneck*. This design inhibits modularity and also creates a security vulnerability whereby parent signal functions have complete access to their children's inputs and outputs. Allowing signal functions themselves to perform effects would alleviate this problem, but it can interfere with the fundamental abstraction. We present the notion of *resource types* to address this issue and allow the fundamental abstraction to hold in the presence of effects. Resource types are phantom type parameters that are added to the type signatures of signal functions that indicate what effects those signal functions are performing and leverage the type-checker to prevent resource

usage that would break the abstraction. We show type judgments and operational semantics for a resource-typed model as well as an implementation of the system in Haskell.

FRP typically relies on a notion of *synchrony*, or the idea that all streams of data are synchronized across time. In fact, this synchrony is a key component of maintaining the fundamental abstraction as it ensures that two disparate portions of the program will receive the same deterministically associated (synchronous) input values and that their separate results will coordinate in the same output values. However, in many applications, this synchrony is too strong. We discuss a notion of treating time not as a global constant that governs the entire program uniformly, but rather as *relative to a given process*. In one process, time will appear to progress at one rate, but in another, time can proceed differently. Although we forfeit the global impact of the fundamental abstraction, this allows us to retain its effects on a per-process scale. That is, we can assume each process processes its inputs continuously despite the whole network having different notions of time. To allow communication between these asynchronous processes, we introduce *wormholes*, which act as specialized connections that apply a sort of *time dilation* to information passing through them. We additionally show that they can be used to subsume other common FRP operations such as looping and causality.

We apply the concepts of all of these ideas into a functional reactive library for graphical user interfaces called UISF. Thus, this work concludes with an overview and examples of practically using our version of FRP.