JFP **27**, e16, 11 pages, 2017. © Cambridge University Press 2017
doi:10.1017/S0956796817000089

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 9 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

GUMSMP: A Scalable Parallel Haskell Implementation

MALAK ALJABRI
University of Glasgow, UK

Date: November 2015; Advisor: Hans-Wolfgang Loidl and Phil Trinder
URL: `http://tinyurl.com/mc4huse`

The most widely available high performance platforms today are hierarchical, with shared memory leaves, e.g. clusters of multi-cores, or NUMA with multiple regions. The Glasgow Haskell Compiler (GHC) provides a number of parallel Haskell implementations targeting different parallel architectures. In particular, GHC-SMP supports shared memory architectures, and GHC-GUM supports distributed memory machines. Both implementations use different, but related, runtime system (RTS) mechanisms and achieve good performance. A specialised RTS for the ubiquitous hierarchical architectures is lacking.

This thesis presents the design, implementation, and evaluation of a new parallel Haskell RTS, GUMSMP, that combines shared and distributed memory mechanisms to exploit hierarchical architectures more effectively. The design evaluates a variety of design choices and aims to efficiently combine scalable distributed memory parallelism, using a virtual shared heap over a hierarchical architecture, with low-overhead shared memory parallelism on shared memory nodes. Key design objectives in realising this system are to prefer local work, and to exploit mostly passive load distribution with pre-fetching.

Systematic performance evaluation shows that the automatic hierarchical load distribution policies must be carefully tuned to obtain good performance. We investigate the impact of several policies including work pre-fetching, favouring inter-node work distribution, and spark segregation with different export and select policies. We present the performance results for GUMSMP, demonstrating good scalability for a set of benchmarks on up to 300 cores. Moreover, our policies provide performance improvements of up to a factor of 1.5 compared to GHC-GUM.

The thesis provides a performance evaluation of distributed and shared heap implementations of parallel Haskell on a state-of-the-art physical shared memory NUMA machine. The evaluation exposes bottlenecks in memory management, which limit scalability beyond 25 cores. We demonstrate that GUMSMP, that combines both distributed and shared heap abstractions, consistently outper- forms the shared memory GHC-SMP on seven benchmarks by a factor of 3.3 on average. Specifically, we show that the best results are obtained when shar- ing memory only within a single NUMA region, and using distributed memory system abstractions across the regions.

Quotient Inductive-Inductive Definitions

GABE DIJKSTRA
University of Nottingham, UK

Date: February 2017; Advisor: Thorsten Altenkirch
URL: http://tinyurl.com/lndmo4k

In this thesis we present a theory of quotient inductive-inductive definitions, which are inductive-inductive definitions extended with constructors for equations. The resulting theory is an improvement over previous treatments of inductive-inductive and indexed inductive definitions in that it unifies and generalises these into a single framework. The framework can also be seen as a first approximation towards a theory of higher inductive types, but done in a set truncated setting.

We give the type of specifications of quotient inductive-inductive definitions mutually with its interpretation as categories of algebras. A categorical characterisation of the induction principle is given and is shown to coincide with the property of being an initial object in the categories of algebras. From the categorical characterisation of induction, we derive a more type theoretic induction principle for our quotient inductive-inductive definitions that looks like the usual induction principles.

The existence of initial objects in the categories of algebras associated to quotient inductive-inductive definitions is established for a class of definitions. This is done by a colimit construction that can be carried out in type theory itself in the presence of natural numbers, sum types and quotients or equivalently, coequalisers.

Certification of Programs with Computational Effects

BURAK EKICI
Université Grenoble Alpes, France

Date: December 2015; Advisor: Jean-Guillaume Dumas and Dominique Duval
URL: `http://tinyurl.com/mre4z9y`

In this thesis, we aim to formalize the effects of a computation. Indeed, most used programming languages involve different sorts of effects: state change, exceptions, input/output, non-determinism, etc. They may bring ease and flexibility to the coding process. However, the problem is to take into account the effects when proving the properties of programs. The major difficulty in such kind of reasoning is the mismatch between the syntax of operations with effects and their interpretation.

Typically, a piece of program with arguments in X that returns a value in Y is not interpreted as a function from X to Y, due to the effects. The best-known algebraic approach to the problem interprets programs including effects with the use of *monads*: the interpretation is a function from X to $T(Y)$ where T is a monad. This approach has been extended to *Lawvere theories* and *algebraic handlers*. Another approach called, the *decorated logic*, provides a sort of equational semantics for reasoning about programs with effects.

We specialize the approach of decorated logic to the state and the exceptions effects by defining *the decorated logic for states* (L_{st}) and *the decorated logic for exceptions* (L_{exc}), respectively. This enables us to prove properties of programs involving such effects. Then, we formalize these logics in Coq and certify the related proofs. These logics are built so as to be sound. In addition, we introduce a relative notion of syntactic completeness of a theory in a given logic with respect to a sublogic. We prove that the decorated theory for the global states as well as two decorated theories for exceptions are syntactically complete relatively to their pure sublogics. These proofs are certified in Coq as applications of our generic frameworks.

On Programming Languages for Probabilistic Modeling

DANIEL E. HUANG
Harvard University, USA

Date: August 2016; Advisor: Greg Morrisett
URL: `http://tinyurl.com/m2qbh3e`

The probabilistic programming paradigm for Bayesian machine learning proposes that we (1) use a programming language to express probabilistic models and (2) leverage the language implementation to perform statistical inference. Notably, these languages typically provide continuous distributions as primitives. Hence, the connection with traditional, discrete notions of computation is not obvious. Nevertheless, there is an intuitive understanding of probabilistic programs in terms of sampling.

In this dissertation, I study probabilistic programming languages, i.e., programming languages designed for probabilistic modeling. In particular, I begin by exploring how (Type-2) computable distributions can be used to give both distributional and (algorithmic) sampling semantics to a high-level, PCF-like language extended with continuous distributions. One motivation for using computable distributions, as opposed to more generally measures, is so that we can think of a Turing-complete probabilistic programming language as expressing computable distributions. Next, I explore how to improve the efficiency of automated inference for the class of fixed-structure, parametric models. Towards this end, I design, implement, and evaluate a language called AugurV2, which proposes probabilistic modeling with parallel comprehensions for composable Markov Chain Monte Carlo (MCMC) inference. Unlike many other probabilistic programming systems, the AugurV2 compiler leverages multiple intermediate languages to successively transform a declarative description of a model into an exectuable inference algorithm. Indeed, I will show how such a compilation strategy enables AugurV2 to scale inference for practical models of interest. I will also show how to use the semantics based on computable distributions to justify certain aspects of AugurV2's design.

Type Theory in a Type Theory with Quotient Inductive Types

AMBRUS KAPOSI
University of Nottingham, UK

Date: November 2016; Advisor: Graham Hutton
URL: `http://tinyurl.com/kjkw76n`

Type theory (with dependent types) was introduced by Per Martin-Löf with the intention of providing a foundation for constructive mathematics. A part of constructive mathematics is type theory itself, hence we should be able to say what type theory is using the formal language of type theory. In addition, metatheoretic properties of type theory such as normalisation should be provable in type theory.

The usual way of defining type theory formally is by starting with an inductive definition of precontexts, pretypes and preterms and as a second step defining a ternary typing relation over these three components. Well-typed terms are those preterms for which there exists a precontext and pretype such that the relation holds. However, if we use the rich metalanguage of type theory to talk about type theory, we can define well-typed terms directly as an inductive family indexed over contexts and types. We believe that this latter approach is closer to the spirit of type theory where objects come intrinsically with their types.

Internalising a type theory with dependent types is challenging because of the mutual definitions of types, terms, substitution of terms and the conversion relation. We use induction induction to express this mutual dependency. Furthermore, to reduce the type-theoretic boilerplate needed for reasoning in the syntax, we encode the conversion relation as the equality type of the syntax. We use equality constructors thus we define the syntax as a quotient inductive type (a special case of higher inductive types from homotopy type theory). We define the syntax of a basic type theory with dependent function space, a base type and a family over the base type as a quotient inductive inductive type.

The definition of the syntax comes with a notion of model and an eliminator: whenever one is able to define a model, the eliminator provides a function from the syntax to the model.

We show that this method of representing type theory is practically feasible by defining a number of models: the standard model, the logical predicate interpretation for parametricity (as a syntactic translation) and the proof-relevant presheaf logical predicate interpretation. By extending the latter with a quote function back into the syntax, we prove normalisation for type theory. This can be seen as a proof of normalisation by evaluation.

Internalising the syntax of type theory is not only of theoretical interest. It opens the possibility of type-theoretic metaprogramming in a type-safe way. This could be used for generic programming in type theory and to implement extensions of type theory which are justified by models such as guarded type theory or homotopy type theory.

Formal Foundations for Hybrid Effect Analysis

YUHENG LONG
Iowa State University, USA

Date: May 2016; Advisor: Hridesh Rajan
URL: `http://tinyurl.com/kmlgjbz`

Type-and-effect systems are a powerful tool for program construction and verification. Type-and-effect systems are useful because it helps reduce bugs in computer programs, enables compiler optimizations and provides program documentation. As software systems increasingly embrace dynamic features and complex modes of compilation, static effect systems have to reconcile over competing goals such as precision, soundness, modularity, and programmer productivity. In this thesis, we propose the idea of combining static and dynamic analysis for effect systems to improve precision and flexibility.

We describe intensional effect polymorphism, a new foundation for effect systems that integrates static and dynamic effect checking. Our system allows the effect of polymorphic code to be intensionally inspected. It supports a highly precise notion of effect polymorphism through a lightweight notion of dynamic typing. When coupled with parametric polymorphism, the powerful system utilizes runtime information to enable precise effect reasoning, while at the same time retains strong type safety guarantees. The technical innovations of our design include a relational notion of effect checking, the use of bounded existential types to capture the subtle interactions between static typing and dynamic typing, and a differential alignment strategy to achieve efficiency in dynamic typing.

We introduce the idea of first-class effects, where the computational effect of an expression can be programmatically reflected, passed around as values, and analyzed at runtime. A broad range of designs "hard-coded" in existing effect-guided analyses can be supported through intuitive programming abstractions. The core technical development is a type system with a couple of features. Our type system provides static guarantees to application-specific effect management properties through refinement types, promoting "correct-by-design" effect-guided programming. Also, our type system computes not only the over-approximation of effects, but also their under-approximation. The duality unifies the common theme of permission vs. obligation in effect reasoning.

Finally, we show the potential benefit of intensional effects by applying it to an event-driven system to obtain safe concurrency. The technical innovations of our system include a novel effect system to soundly approximate the dynamism introduced by runtime handler registration, a static analysis to precompute the effects and a dynamic analysis that uses the precomputed effects to improve concurrency. Our design simplifies modular concurrency reasoning and avoids concurrency hazards.

Context-Aware Programming Languages

TOMAS PETRICEK
University of Cambridge, UK

Date: March 2017; Advisor: Alan Mycroft
URL: `http://tinyurl.com/k2hhjso`

The development of programming languages needs to reflect important changes in the way programs execute. In recent years, this has included the development of parallel programming models (in reaction to the multi-core revolution) or improvements in data access technologies. This thesis is a response to another such revolution – the diversification of devices and systems where programs run.

The key point made by this thesis is the realization that an execution environment or a context is fundamental for writing modern applications and that programming languages should provide abstractions for programming with context and verifying how it is accessed.

We identify a number of program properties that were not connected before, but model some notion of context. Our examples include tracking different execution platforms (and their versions) in cross-platform development, resources available in different execution environments (e.g. GPS sensor on a phone and database on the server), but also more traditional notions such as variable usage (e.g. in liveness analysis and linear logics) or past values in stream-based dataflow programming. Our first contribution is the discovery of the connection between the above examples and their novel presentation in the form of calculi (coeffect systems). The presented type systems and formal semantics highlight the relationship between different notions of context.

Our second contribution is the definition of two unified coeffect calculi that capture the common structure of the examples. In particular, our flat coeffect calculus models languages with contextual properties of the execution environment and our structural coeffect calculus models languages where the contextual properties are attached to the variable usage. We define the semantics of the calculi in terms of category theoretical structure of an indexed comonad (based on dualisation of the well-known monad structure), use it to define operational semantics and prove type safety of the calculi.

Our third contribution is a novel presentation of our work in the form of web-based interactive essay. This provides a simple implementation of three context-aware programming languages and lets the reader write and run simple context-aware programs, but also explore the theory behind the implementation including the typing derivation and semantics.

Stream Processing using Grammars and Regular Expressions

ULRIK TERP RASMUSSEN
University of Copenhagen, Denmark

Date: January 2017; Advisor: Fritz Henglein
URL: `http://tinyurl.com/n2vcy6k`

In this dissertation we study regular expression based parsing and the use of grammatical specifications for the synthesis of fast, streaming string-processing programs.

In the first part we develop two linear-time algorithms for regular expression based parsing with Perl-style greedy disambiguation. The first algorithm operates in two passes in a semi-streaming fashion, using a constant amount of working memory and an auxiliary tape storage which is written in the first pass and consumed by the second. The second algorithm is a single-pass and optimally streaming algorithm which outputs as much of the parse tree as is semantically possible based on the input prefix read so far, and resorts to buffering as many symbols as is required to resolve the next choice. Optimality is obtained by performing a PSPACE-complete pre-analysis on the regular expression.

In the second part we present Kleenex, a language for expressing high-performance streaming string processing programs as regular grammars with embedded semantic actions, and its compilation to streaming string transducers with worst-case linear-time performance. Its underlying theory is based on transducer decomposition into oracle and action machines, and a finite-state specialization of the streaming parsing algorithm presented in the first part. In the second part we also develop a new linear-time streaming parsing algorithm for parsing expression grammars (PEG) which generalizes the regular grammars of Kleenex. The algorithm is based on a bottom-up tabulation algorithm reformulated using least fixed points and evaluated using an instance of the chaotic iteration scheme by Cousot and Cousot.

Combining Type Checking with Model Checking for System Verification

ZHIQIANG REN
Boston University, USA

Date: January 2017; Advisor: Hongwei Xi
URL: `http://tinyurl.com/lw9dhgj`

Type checking is widely used in mainstream programming languages to detect programming errors at compile time. Model checking is gaining popularity as an automated technique for systematically analyzing behaviors of systems. My research focuses on combining these two software verification techniques synergically into one platform for the creation of correct models for software designs.

This thesis describes two modeling languages ATS/PML and ATS/Veri that inherit the advanced type system from an existing programming language ATS, in which both dependent types of Dependent ML style and linear types are supported. A detailed discussion is given for the usage of advanced types to detect modeling errors at the stage of model construction. Going further, various modeling primitives with well-designed types are introduced into my modeling languages to facilitate a synergic combination of type checking with model checking.

The semantics of ATS/PML is designed to be directly rooted in a well-known modeling language PROMELA. Rules for translation from ATS/PML to PROMELA are designed and a compiler is developed accordingly so that the SPIN model checker can be readily employed to perform checking on models constructed in ATS/PML. ATS/Veri is designed to be a modeling language, which allows a programmer to construct models for real-world multi-threaded software applications in the same way as writing a functional program with support for synchronization, communication, and scheduling among threads. Semantics of ATS/Veri is formally defined for the development of corresponding model checkers and a compiler is built to translate ATS/Veri into CSP# and exploit the state-of-the-art verification platform PAT for model checking ATS/Veri models. The correctness of such a transformational approach is illustrated based on the semantics of ATS/Veri and CSP#.

In summary, the primary contribution of this thesis lies in the creation of a family of modeling languages with highly expressive types for modeling concurrent software systems as well as the related platform supporting verification via model checking. As such, we can combine type checking and model checking synergically to ensure software correctness with high confidence.