Research Summary and Prospectus

Patricia Johann

1. Introduction

My research is part of the field of study known as *theoretical computer science*, a confluence of mathematics and computer science which aims to provide a solid mathematical foundation for real-world computing. Within the realm of theoretical computer science, my work has focused on applying transformation-based methods to increase the efficiency of first- and higher-order resolution theorem provers, and, for the past four or five years, programs written in strategy-based and functional programming languages. It heartily embraces the point of view that program transformations can provide a solid practical and theoretical basis for developing efficient automatable solutions to computational problems.

My interest in computation derives from the observation that proofs of properties of formal logics often carry computational content. My research over the last two decades has been along two major threads related to this observation. First, logics can give rise to calculi which provide means of automating deduction in the logics they describe. My graduate and initial postdoctoral years therefore found me concerned with automatic theorem proving in a sequence of increasingly expressive calculi, and particularly with developing transformation-based algorithms for solving unification problems in them. Unification is a pattern-matching computation which can be interpreted as imposing constraints on proof steps in deductions. Considering unification problems from this point of view soon led to my interest in algorithmic solutions to more general constraint problems which arise in the context of automated deduction. It also led to my work on the well-founded orderings from which such constraints are typically derived.

Secondly, over the past decade I have become interested in mathematically-based computational techniques other than those which are the mainstay of traditional automatic theorem proving. As a result, most of my research now focuses on the mathematical foundations of programming languages. I have investigated reasoning principles for macro languages, as well as the theory and the practice of transformation-based program optimization methods for functional languages and strategy-based languages. Optimizing transformations take programs to be optimized as input and return their optimized equivalents; since such transformations can be implemented as programs, they can be described both playfully and accurately as "programs that make other programs run faster." Automatic program transformation enables programmers to work at a higher level of abstraction than would otherwise be possible, and this, in turn, gives rise to increases in programmer productivity. It can also lead to improved reliability, maintainability, reusability, and efficiency of software. Gains in these areas are instrumental to recovering some of the \$59.5 billion that software errors are estimated cost the U.S. economy annually [1].

The theory and practice of program optimization ties squarely into the issue of parametricity and its myriad consequences. My desire to understand the deep nature of parametricity has informed most of my research over the past decade and continues to do so to this day. Parametricity constrains the behavior of polymorphic functional programs in a way that allows the derivation of interesting theorems about them solely from their types. These theorems arise from the fact that a parametric polymorphic function must behave uniformly, i.e., must use the same algorithm to compute its result regardless of the concrete type at which it is instantiated. Interestingly, this uniformity can be used not only to derive optimizing transformations, but also to prove that they do not otherwise change the computational meanings of programs to which they are applied. The main obstacle to understanding parametricity either in general or for a particular programming language in question, is that formalizing our intuitions about it is notoriously difficult.

This document summarizes my research and publications in both automated deduction and programming languages. Overall, I have one graduate-level textbook and 22 internationally refereed publications, including 11 journal papers, 10 conference papers, and one paper in an electronic journal (see accompanying CV). Four more submissions are currently undergoing similar review for journal publication now, and one is undergoing review for presentation at a prestigious international conference in programming languages. All of my published papers, as well as these five submissions, are included in this dossier. As detailed below, two of my recent papers overturn the conventional wisdom on their respective topics, and the submission on GADTs included in this dossier gives these cutting-edge data types their first initial algebra semantics. My research has been supported five separate, highly-competitive awards from the National Science Foundation (NSF), as well as by grants from the German Academic Exchange Service (DAAD), Bates College, and Rutgers University. These grants total just shy of \$400,000. Three additional travel and curriculum development awards raise the total to about \$420,000.

This summary is divided into two main sections. The first section describes my research in automated deduction, while the second section describes my research in programming languages. The summary is organized in, roughly, chronological order. My work in automated deduction was undertaken primarily at Wesleyan University, Hobart and William Smith Colleges (HWS), and Universität des Saarlandes. I finished my work in this area at the Oregon Graduate Institute (OGI), where I took a research position in 1995 with the aim of completing a graduate-level textbook on deduction systems, and then transitioning my research to the area of programming languages. Being trained as a mathematician in a mathematics department, and having had only minimal exposure to computer science by virtue of having come to it through the "back door" of the lambda calculus, I lacked the traditional background in the discipline. But I had heard that programming languages could be built on solid mathematical foundations, and I found this idea positively fascinating. Many years later, I am pleased to report that the transition to programming languages research was well worth the effort, and that I derive a great deal of satisfaction from the kind of work I now do. As an added bonus, I quite happily find my training in mathematics to be a significant asset in this line of work.

With the exception of that carried out at the Universität des Saarlandes and OGI, all of the research described in this summary has been performed at small liberal arts colleges (HWS, Dickinson College, and Bates) or at Rutgers-Camden. Each of these institutions requires faculty to teach, each academic year, the equivalent of either five or six (depending on the institution) one-semester courses. At none of these institutions have I had colleagues also conducting research in programming languages or automated deduction; in fact, at Bates I was the only computer scientist at all at the College, and was hired specifically to be in charge of the College's entire computer science program (administered by the mathematics department, of which I was a member). Throughout my career, then, teaching has constituted a significant portion of my responsibilities, and my research has been conducted without the benefit of local collaboration. In the absence of master's or PhD students at any of these institutions (an absence which has likely caused my grant income lower than it would otherwise have been, since one cannot apply for full-time support of undergraduates), I have tried to involve undergraduates in my research to the greatest extent possible. Overall, I have guided 9 undergraduate summer research projects, 9 independent studies and undergraduate academic year research projects, and one honors thesis. Of course, involving undergraduates in one's research rarely does much to advance the work; in fact, it usually slows research progress. But it does help ensure that research ideas filter down to students to some degree, and thus has the opportunity to influence how at least some of them think about computer science. On the other hand, because undergraduate training is, in my experience, more properly regarded as a teaching activity than as a research activity, it is discussed in detail in the section on teaching in the cover letter included in this dossier rather than in this summary.

2. Automated Deduction

My early research focused on applying transformation-based methods to increase the efficiency of first- and higher-order resolution theorem provers. A significant portion of my more recent work carries the transformation-based aspect of this research thread forward, focusing as it does on transformation-based techniques for improving the efficiency of programs written in functional and strategy-based programming languages. Indeed, much of the research I have done over the years heartily embraces the point of view that transformation-based methods can be fruitful in developing efficient solutions to computational problems.

The specific problems in automated deduction on which I have worked are:

- *First- and Higher-Order Equational Unification*

  My postgraduate and initial postdoctoral years found me concerned with automatic theorem proving in a sequence of increasingly expressive calculi, and focusing particularly on developing transformation-based unification algorithms for them. The conference and journal versions of "An Improved General

E-unification Method," written together with my PhD advisor, Dan Dougherty, developed an efficiency improvement for the then-state-of-the-art transformation-based technique developed by Jean Gallier and Wayne Snyder in 1989 for unifying first-order terms modulo an arbitrary equational theory. This was accomplished by building the equational theory into the unification algorithm in a way which more delicately simulates the completion of non-convergent theories. The conference and journal versions of "A Combinatory Logic Approach to Higher-order E-unification" also written with Dougherty, showed how to extend his then-recent recasting of higher-order unification modulo $\beta\eta$-equality in terms of simply typed combinatory logic to accommodate any first-order equational theory admitting presentation as a convergent term rewriting system. This work required a complete redevelopment of Curry's theory of combinatory normal forms in the presence of a convergent first-order theory. Such a redevelopment appeared in my paper "Normal Forms in Combinatory Logic."

- *Order-Sorted Unification*

  The incorporation of sort information into first-order logic is well-known to reduce search spaces associated with automated deduction in those logics, and holds considerable promise for similarly improving automated deduction in higher-order logics as well. At the Universität des Saarlandes I was a member of the research group of Prof. Dr. Jörg Siekmann, where I worked with Michael Kohlhase to develop order-sorted higher-order calculi and transformation-based unification algorithms for higher-order calculi (both lambda calculus and combinatory logic variants) for use in the order-sorted higher-order resolution theorem prover Ω. This work is reported in "Unification in an Extensional Lambda Calculus with Ordered Function Sorts and Constant Overloading" written together with Kohlhase, and in my technical report "A Combinator-based Order-sorted Higher-order Unification Algorithm." This research was supported by the DAAD and the NSF.

  In addition, to help techniques in first-order order-sorted resolution theorem proving become part of the standard automated deduction repertoire, I co-authored the graduate-level textbook *Deduction Systems* with Rolf Socher-Ambrosius. This book provided the first-ever textbook treatment of resolution theorem proving in an order-sorted logic. It also offers a mathematically rigorous treatment of the fundamental concepts and methods of first-order resolution theorem proving.

- *Simplification Orderings*

  The kinds of sort constraints described above induce partial orderings on terms which can be used to increase the efficiency of deduction engines. But both theoretical results and experiments with existing theorem provers also confirm the advantages of deduction methods constrained by more general well-founded orderings on terms, such as lexicographic path orderings (LPOs), recursive path orderings (RPOs), and other simplification orderings (SOs). To determine whether or not admissible ordered deduction inference steps can be carried out, techniques for solving the relevant ordering constraints are needed. In "Solving Simplification Ordering Constraints," Socher-Ambrosius and I gave a simple polynomial-time transformation-based procedure for deciding satisfiability of ordering constraints by SOs, and showed that the corresponding problem for total SOs is NP-complete. This latter result can be interpreted as showing that the problem of deciding whether or not a simplification ordering on first-order terms can be linearized is NP-complete, and has as a corollary that the problem of deciding satisfiability of ordering constraints by LPOs is NP-complete. This work was supported by the NSF.

3. Programming Languages

Since my move from automated deduction to programming languages eleven years ago, the overarching question around which my research has revolved is this:

What is the nature of parametricity, and what are its consequences for practical programming in various computational settings?

This question is multifaceted enough to provide a unifying contextual backbone for my research, on the one hand, and to give rise to a variety of specific research questions which help shed light on the larger issue, on the other. Each of the approaches to parametricity and its consequences that I have considered — domain-theoretic, operational, and categorical — has offered valuable insights. An especially important (to me) aspect of my research in this area is that it takes a rigorous, mathematical, foundational approach.

- *Correctness of Short Cut Fusion and of Its Generalizations for Algebraic Data Types*

  I first got interested in parametricity and its consequences when I joined OGI. Early on in my time there, I read the paper "A Short Cut to Deforestation" [3] by Andrew Gill, John Launchbury and Simon Peyton Jones describing short cut fusion. *Fusion* (also known as deforestation) is the process of improving the efficiency of modularly constructed functional programs by transforming them into monolithic equivalents. This is accomplished by eliminating from them so-called *intermediate data structures*, i.e., data structures which play no computational role in a modular program other than to "glue" together its components. *Short cut fusion* uses a single, local program transformation rule — called the `foldr`/`build` rule — to fuse compositions of list-processing functions via applications of traditional `fold`/`unfold` program transformation steps. I wondered how one could actually prove that short cut fusion is correct, in the sense of preserving the computational meanings of programs.

  One way to do this involves constructing a parametric model for the underlying language which respects the computational meanings of its programs. A *parametric model* for a language is one whose notion of equivalence is induced by a Reynolds-style logical relation for that language. That the polymorphic functions of a language admitting a parametric model are "data type independent" follows from the observation that every such function satisfies the parametricity theorem for its type, i.e., is related to itself by the relational interpretation of its type. Cleverly instantiating the parametricity theorems for various types gives a variety of so-called *free theorems*. The fusion rules in which we are interested are precisely such free theorems, and thus must hold in any parametric model of the underlying language. Since computational meanings of programs can be given by operational semantics, it is particularly useful to construct parametric models for which the notion of equivalence induced by the logical relation on the underlying language coincides with observational equivalence of its programs. Such *parametric models of observational equivalence* can be used to prove that applications of short cut fusion and related transformations do not change the observable behavior of programs.

  In 2000, Andrew Pitts showed how to construct a parametric model of observational equivalence for PolyPCF, an extension of the Girard-Reynolds lambda calculus $\lambda^{\forall}$ with fixpoints and list data types, by building the operational semantics directly into the logical relation. In "Short Cut Fusion is Correct" I used Pitts' model to give the first-ever formal proof of the correctness of short cut fusion (in this case, for PolyPCF). In "A Generalization of Short Cut Fusion and Its Correctness Proof," I extended Gill's `foldr`/`augment` generalization of the `foldr`/`build` rule for lists to give analogous fusion rules for arbitrary algebraic data types. These `fold`/`augment` rules can be thought of as optimizing compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of them. I also used Pitts' parametric model of observational equivalence for PolyFix (PolyPCF extended with non-list algebraic data types) to prove the correctness of these rules for that calculus. So "A Generalization of Short Cut Fusion and Its Correctness Proof" generalizes "Short Cut Fusion is Correct" along both the fusion rules and the data types considered.

  "On Proving the Correctness of Program Transformations Based on Free Theorems for Higher-order Polymorphic Calculi" is the last in this series of papers. In it, I raise and clarify some of the delicate but fundamental issues that must be addressed when constructing correctness proofs for program transformations — including, but not limited to, short cut fusion and its generalizations — that are derivable as free theorems. These issues typically go unaddressed in correctness proofs appearing in the literature; as a result, most such proofs are incomplete, and most free theorems-based transformations are applied to programs in calculi for which they are not actually known to be correct. I also offer a principled approach to developing such proofs. Finally, I show how Pitts' techniques can be used to prove the correctness, for PolyFix, of transformations based on the Acid Rain theorems of Akihiko Takano and Erik Meijer. Correctness for PolyFix of the `fold`/`build` rule, the `destroy`/`unfoldr` rule which is its dual, the `fold`/`augment` rule which generalizes it, and the hylofusion rules of Takano and

Meijer all follow immediately. In work with Janis Voigtländer discussed below we show how these same techniques can be extended to construct parametric models of observational equivalence — and thus to derive complete correctness proofs for free theorems-based transformations — for calculi more closely resembling languages with which programmers are concerned in practice.

- *Fusion and Strategy-based Languages*

  I have also been interested in the development and implementation of fusion techniques for improving the performance of programs in strategy-based languages such as Eelco Visser's Stratego. Stratego supports a strong separation between the logic of rewrite rules and the strategies which control their application, and this makes it possible to configure different algorithms by combining problem-specific transformation rules with reusable generic application strategies. In "Fusing Logic and Control with Local Transformations: An Example Optimization,", Visser and I develop, and implement in Stratego itself, a fusion technique for Stratego's generic innermost reduction strategy. Fusion techniques for strategic programs are naturally based on traversal schemes rather than on the structure of data types; ours specializes the innermost reduction strategy to any given set of rules. Rutgers student Jon Pospischil's 2005 honors thesis developed an analogue of this fusion technique for Stratego's generic outermost reduction strategy. Fusion with both the innermost and outermost strategies help support abstract strategic programming without sacrificing the efficiency of hand-written Stratego specializations. The work on fusion with the outermost strategy was supported by Rutgers University.

  In "Warm Fusion in Stratego: A Case Study in Generation of Program Transformation Systems," Visser and I turn the tables slightly and apply Stratego to the problem of fusing functional programs. Specifically, we program the warm fusion algorithm of John Launchbury and Tim Sheard in Stratego. *Warm fusion* is a technique for converting a program written in the general recursive style into a program to which short cut fusion can immediately be applied; this is achieved by combining the `fold` promotion techniques of Tim Sheard and Leonidas Fegaras with a generalization of the technique of Peyton Jones and Launchbury for splitting a function into so-called worker-wrapper form. This case study provides experience with the design and implementation, in Stratego, of a complete transformation system, including interfaces with a parsing and typechecking front-end and a pretty-printing back-end for Haskell. It also establishes for the first time that the warm fusion technique is fully automatable. Indeed, as mentioned above, Stratego's separation of logic from control ensures that both the design and implementation of warm fusion are thoroughly modular, making it easy to modify the set of program transformation rules, for example, as well as to experiment with a variety of strategies for applying rules. It was precisely this kind of experimentation that led to the "double splitting" wrapper-worker technique introduced in the paper for recognizing certain variables as static parameters of programs undergoing warm fusion, and thus to making explicit a transformation step which happens "automagically" in the original paper of Launchbury and Sheard.

- *Parametricity for More Realistic Languages*

  A good deal of current research has been concerned with understanding parametricity properties of, and constructing parametric models of observational equivalence for, calculi which resemble modern functional programming languages. Such models can be used to prove the correctness of short cut fusion and other program transformations based on free theorems and thus, ultimately, on parametricity. In "Free Theorems in the Presence of *seq*," Janis Voigtländer and I construct a parametric model for PolyPCF augmented with Haskell's built-in selective strictness operator *seq* — which allows programmers to override Haskell's default 'lazy' evaluation order — and use it to investigate the impact of selective strictness on parametricity. In particular, we show that the decades-old conventional wisdom about the ways in which selective strictness breaks standard parametricity results is, surprisingly, incorrect. This conventional wisdom stated that the interpretations of types by the underlying logical relation — and thus that all functions in free theorems, where one has a choice — must be admissible and bottom-reflecting. But we give counterexamples showing that these restrictions aren't sufficient to recover standard parametricity results when *seq* is present. We also identify restrictions that *are* sufficient, capturing the asymmetry in program termination behavior induced by *seq* in modified relational interpretations of types, and using the resulting logical relation to derive 'inequational' free theorems which hold even in the presence of *seq*. While the move from the equational to the inequational setting might appear at first to be an unnecessary weakening of parametricity results, it actually adds important strength: we obtain *more* free theorems than would be possible via a less radical revision of the standard equational approach — including ones that have no valid equational counterpart when *seq* is present — and equational free theorems can often be recovered by combining two inequational ones. Moreover, our inequational free theorems are useful from a practical, as well as from a theoretical, point of view. Although it is natural to insist on the correctness of program transformations whenever possible, in some circumstances provable correctness may be more than is actually necessary. Indeed, in applications it is often enough just to know that the programs produced by a transformation are always at least as defined as the original ones. A transformation may thus be useful in practice even if can only be proved to be observationally safe — in the sense just described — rather than fully correct.

Finally, we consider in detail the application of our results to short cut fusion and related program transformations, and derive restrictions that can be imposed on the functions appearing in these transformation rules to ensure their continued correctness even when *seq* is present. The journal paper "The Impact of *seq* on Free Theorems-Based Program Transformations" expands "Free Theorems in the Presence of *seq*" to include proofs that were omitted from the latter due to space limitations, and to generalize slightly the results there about the `destroy/unfold` rule in the latter. It also shows that the restrictions imposed to recover free theorems in the presence of *seq* cannot be weakened, and discusses the potential implications of our results on parametricity for purely strict languages. The inconsistency between the practical utility of *seq* and its incompatibility with standard parametricity properties that this work resolves is discussed in Section 10.3 of "A History of Haskell: Being Lazy with Class" by Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler [2].

The construction just described of a parametric model for PolyPCF with *seq* was carried out relative to the standardly accepted naive denotational semantics for Haskell. This is primarily because working in a denotational setting allowed a more intuitive initial approach to the problem. But in "Selective Strictness and Parametricity in Structural Operational Semantics, Inequationally," Voigtländer and I combine Pitts' operational techniques for constructing parametric models of observational equivalence with our denotationally-based results about recovering parametricity in the presence of *seq* to arrive at a parametric model of observational approximation for PolyPCF with *seq*. This model is appropriate as a formal basis for deriving new 'inequational' parametricity results, as well as proving 'approximation correctness' results for short cut fusion and related program transformations, all of which hold when *seq* is present. Our construction of this model is similar to Pitts' original one for PolyPCF, but has been refined to accommodate both inequational reasoning and the restrictions on relational interpretations of types imposed by *seq*. While these restrictions are in some sense 'just' operational analogues of those in the denotational setting, their translation and incorporation into the operationally-based logical relation was accomplished rather differently from what we had imagined when writing those earlier papers. Indeed, while we had originally thought we'd have to build operational analogues of both admissibility and totality directly into the logical relation, preservation of the latter by the former ensures that only the former need be built in directly and the latter can be layered modularly on top. Another difference between our construction and that of Pitts is that ours starts from a small-step semantics rather than from a big-step semantics. This allows us to more clearly reflect the operational behavior of *seq*, and it simultaneously provides new insights into techniques for modularly constructing parametric models of operational equivalence and approximation for extensions of PolyPCF (or PolyFix) with various language features.

The ultimate goal of the line of research advanced in my papers with Voigtländer is the development of tools for reasoning about parametricity properties of, and free theorems-based transformations on programs in, real programming languages rather than toy calculi. The operational approach taken in "Selective Strictness and Parametricity in Structural Operational Semantics, Inequationally" is beneficial in this regard. Indeed, while the Glasgow Haskell Compiler (GHC) uses a variant of $\lambda^{\forall}$ as its intermediate language Core, a well-defined denotational semantics is not currently known even for relevant subsets of Core. It is thus currently unclear whether results derived relative to any particular denotational model of, say, PolyPCF with *seq* would eventually shed any light at all on parametricity properties of Core. On the other hand, Voigtländer and I derive our results relative to an operational semantics very much like the one that implementations like GHC are expected to satisfy, and so our results do indeed provide insights into parametricity properties of Core and, by extension, full Haskell.

In "A Family of Logical Relations for the Semantics of Haskell," Voigtländer and I show how to move the discussion of parametricity and free theorems-based program transformations even more toward real languages by deriving parametricity results for a calculus which supports not only selective strictness (now coded as a strict case construct because calls to *seq* in Haskell are translated into such constructs in Haskell's Core), but also includes an explicit error primitive capable of distinguishing between different kinds of computational failure (e.g., divergence versus failed pattern matching). Distinguishing different failure causes is important in practice, because conflating them means that a program transformation that is claimed to be semantics-preserving may very well transform a nonterminating program into one that instead terminates with a runtime error, or vice versa, or may confuse different kinds of runtime errors; Haskell examples are easily given for which the `fold`/`build` rule transforms arbitrary different failures into one another in this way. Once distinguished, the relative definedness of different failure causes needs to be considered, because different orders here induce different observational relations on programs. So rather than developing a single new logical relation, we instead develop an entire family of logical relations parameterized over a definedness order on failure causes, each member of which characterizes a corresponding observational relation. Significantly, the abstract definedness order with which we work has both observational equivalence and observational approximation for our calculus among the specializations it induces.

Although we deal with properties very much tied to types, we base our results on a type-erasing semantics since such a semantics is more faithful to actual implementations. The choice of whether to leave type generalization and specialization implicit (as in Haskell) or to make them explicit (as in $\lambda^{\forall}$ and its extensions) may at first glance seem to be just an implementation detail, but it can actually change which program termination behaviors of terms of polymorphic type are observable (whether or not selective strictness is supported). Parametricity results must therefore be derived relative to a logical relation which has been constructed appropriately for the intended program semantics. Haskell's implicit type generalization and specialization require that we work with a semantics defined on the type-erasures of terms. Persisting in working instead with a typeful semantics such as the one in "Selective Strictness and Parametricity in Structural Operational Semantics, Inequationally" imposes extra convergence conditions on polymorphic arguments in parametricity results — conditions which were not found to be necessary on the primarily intuitive level in either "Free Theorems in the Presence of *seq*" or "The Impact of *seq* on Free Theorems-Based Program Transformations", and indeed are not justifiable with respect to the semantics of Haskell. Working with a type-erasing semantics thus provides a foundational treatment of parametricity which is more in line with Haskell's Core than previous ones.

The general approach we take in constructing the family of logical relations in "A Family of Logical Relations for the Semantics of Haskell," is similar to Pitts', but it is not obvious *a priori* that his machinery can be brought to bear there; indeed, the additional language features must be handled appropriately, and the interaction between the low-level semantics on the untyped level and the intended reasoning on the higher, typed level poses a particular challenge. On the other hand, parameterizing our family of logical relations allows us to deal with semantic equivalence and (either direction of) semantic approximation in a unified manner. By contrast, previous work on logical relations for polymorphic languages has dealt exclusively with either an equational setting or an inequational one, and this has led to a certain repetition in proofs of key results and their applications. Such repetition can be avoided using our techniques, since a single proof parameterized by an abstract preorder suffices, and results for equivalence and different choices of approximation can then be read off by just instantiating this parameter appropriately. Our parameterized approach is also easily extended to accommodate additional language features. We illustrate this by showing how existential types, which are not yet present in Haskell, can be integrated into our framework *a posteriori* while preserving parametricity.

This entire body of work has been supported by the National Science Foundation, and my collaboration with Voigtländer is ongoing. One specific direction for future research involves devising a qualified type system for '*seq*ability' along the lines of that of John Launchbury and Ross Paterson for pointedness. This should provide finer control over the impact of *seq* on programs, and so allow the derivation of free theorems under less restrictive conditions than is possible with the more global analysis we have considered thus far. Another topic for future investigation is to consider parametricity for the dual of PolyPCF with *seq*, i.e., for a default strict language with explicit laziness primitive. Work in this area will, of course, build on the many recent papers on parametricity in purely strict languages.

- *Fusion for Advanced Data Types*

  The strand of my current research about which I am most excited focuses on deriving initial algebra semantics for advanced data types. As described in more detail below, this strand ties very nicely into the work on parametricity and free theorems-based program transformations just discussed. In particular, `fold` and `build` combinators, and a `fold/build` fusion rule, can be derived for any data type admitting an initial algebra semantics. In fact, an initial algebra semantics also gives a so-called *Church encoding* of a data type into $\lambda^{\forall}$. Having a `fold` combinator, a `build` combinator, a Church encoding, and a `fold/build` rule — which, collectively, I call an *initial algebra package* of programming tools — for a data type ensures that structured programming with data of that type, and principled reasoning about programs involving that data, is possible.

  In "Monadic Augment and Generalised Short Cut Fusion," Neil Ghani, Tarmo Uustalu, Varmo Vene, and I explore the category-theoretic ideas first presented in Ghani, Uustalu, and Vene's "Build, Augment, and Destroy. Universally." from a functional programming perspective. This paper showed how to derive an initial algebra package for every inductive type, i.e., every type which is the least fixed point of some (covariant) functor. It takes as its point of departure the category-theoretic interpretation of algebraic data types as least fixed points of functors, and extends the resulting initial algebra semantics for algebraic types to *all* inductive types. This puts even non-algebraic inductive types on exactly the same solid mathematical foundation as algebraic data types, and makes it possible to program with them in exactly the same structured ways that are well-understood and commonly used for algebraic data types.

  This paper also paves the way for a more general theory of fusion. It asks whether `augment` combinators and `fold/augment` rules can be defined for all inductive types and answers this question in the negative. But it also shows that for a large class of types — namely, those which are fixed points of parameterized monads, and thus are *both* monads *and* inductive types — these constructs can indeed be defined, and can even be defined generically. The generic approach makes it possible to give the first-ever `augment` combinators and `fold/augment` rules for some algebraic data types, such as rose trees, which are commonly used in functional programming but for which such combinators and fusion rules were not previously known to exist. It also makes it possible to define for the first time `augment` combinators and `fold/augment` fusion rules for data types, such as hyperfunctions, which are fixed points of parameterized monads but not algebraic data types. And when specialized to types for which `augment` combinators were already known, the generic approach yields more expressive ones. A particularly compelling aspect of this work is that it shows `augment` to be essentially monadic in nature by establishing that it is interdefinable with `bind`. As a result, the `fold/augment` rules it yields are arguably the most generally applicable fusion rules obtainable for monadic types, and `augment` can be seen as a "fusion-optimized" version of `bind`. Moreover, since the `bind` operation for a fixed point of a parameterized monad can be written as a call to its `fold` combinator followed by a call to its `augment` combinator, these `fold/augment` fusion rules can be applied whenever an application of `bind` is followed by a `fold`, as well as whenever `bind`s occur in sequences of the form `(...((m 'bind' k1) 'bind' k2)... 'bind' kn)`. These computational patterns are expected to occur often, since the `bind` operation is the fundamental operation in monadic computation. These fusion rules are thus expected to be widely applicable.

  The journal version of "Monadic Augment and Generalised Short Cut Fusion," by Ghani and me, constitutes a substantial expansion of the ICFP version, including more expository material to aid the reader, deeper and more careful explanations of the kind afforded by the journal format, implementations of new nontrivial examples of a more sophisticated nature than appear in the conference version, and indicative benchmarking demonstrating roughly a 10% gain in program efficiency. It also includes insights not present in the ICFP version. For example, it more fully explores the idea of an *algebra of fusion*, shows how further generic fusion rules can be derived from the `fold`/`augment` rules for inductive types, and illustrates this idea by deriving entirely new generic `fold`/`map` and `map`/`build` fusion rules for least fixed points of arbitrary bifunctors (parameterized monads are special bifunctors).

In "Initial Algebra Semantics is Enough!," Ghani and I continue this line of research, giving initial algebra packages for nested types. These are based on the representation of nested types as least fixed points of higher-order functors, which gives rise to what we might call the *standard initial algebra semantics* for nested types. This semantics has been known for some time but, unfortunately, the `fold` combinators derived from it have long been thought to be too inexpressive to capture the kinds of programming people want to do with nested types. This consideration led to the development of so-called *generalized* `fold`*s* by Richard Bird and Ross Paterson, together with a whole line of papers on practical programming with them, many of these by Ralf Hinze. Although generalized `fold`s are sufficiently expressive for programming with nested types, they are not, in general, true `fold`s. As a result, no principled approach to programming with, or reasoning about, generalized `fold`s has been known. Ghani and I use right Kan extensions to show that the standard `fold` for each nested type — i.e., the `fold` combinator derived from the standard initial algebra semantics for that type — is interdefinable with its generalized `fold`. This result can be interpreted as showing that generalized `fold`s are merely syntactic sugar for the corresponding standard `fold`s. This is significant because true `fold`s, i.e., about `fold`s obtained from initial algebra semantics, are very well-understood. Our results thus restore initial algebra semantics for nested types to its former glory.

Having shown that standard `fold`s are useful for practical programming with nested types, we are naturally led to consider whether or not Church encodings, corresponding standard `build` combinators, and standard `fold`/`build` rules can be defined for nested types. We show that, like standard `fold`s, these are all derivable uniformly over nested types, and thus give rise to a single version of each standard construct which can be instantiated to any particular nested type of interest. We also use the standard initial algebra semantics for nested types, together with left Kan extensions, to give the first generalized `build`s and generalized `fold`/`build` rules for these types. These generalized constructs correspond to the generalized `fold`s in the literature just as the standard constructs correspond to the standard `fold`s, and so comprise a useful package of tools for programmers who prefer to program with the generalized constructs rather than their corresponding standard ones. Moreover, in precisely the same way that the standard `fold`s and generalized `fold`s are interdefinable, so the other generalized constructs are each interdefinable with their standard counterparts. In fact, as with the standard constructs, the derivation of each of the generalized constructs is uniform over nested types, and so we actually get one generalized version of each construct, each of which can be instantiated to any particular nested type of interest, and each of which is interdefinable with its corresponding standard construct just as each of its instantiations is. Similar results obtain for `destroy` and `unfold` combinators for nested types in the coinductive setting. In "Programming with Nested Types: A Principled Approach," Ghani and I expand "Initial Algebra Semantics is Enough!" to include more background material, a significantly deeper and more self-contained discussion of the category-theoretic basis of our results, and both more, and more substantive, examples.

My most recent work along these lines, also with Ghani, gives initial algebra semantics — and thus initial algebra packages — for GADTs. Our paper "Foundations of Structured Programming with GADTs" was just submitted to a major conference in July. Like all of this work in this research strand, the results described here have been implemented in (an extension of) Haskell, and thus come with a partial correctness guarantee. This work constitutes the first phase of research on GADTs that has been funded by the NSF starting in August 2007.

Ghani and I originally thought we'd derive initial algebra packages for (covariant) GADTs by first showing that every GADT is equivalent to a nested type, then deriving an initial algebra package for that nested type using the techniques in "Initial Algebra Semantics is Enough!," and, finally, deriving from the initial algebra package for the derived nested type a corresponding one for the original GADT. A direct implementation of this approach fails, however, because most such GADTs do not support the necessary functorial action on morphisms in the category of types — i.e., do not support an `fmap` operation, in Haskell parlance — and so *most GADTs are not functors* in the sense required to represent them as nested types. On the other hand, every GADT does have a functorial action on identity morphisms, and so can be considered a functor of kind `|*| -> *`, where `|*|` is the discrete category, i.e., essentially the *set*, whose objects are the types in `*`. Thus, with some slight modifications to reflect the change in domain category, the same techniques used in "Initial Algebra Semantics is Enough!" to give initial algebra semantics and packages for nested types can be used to give initial algebra semantics and packages for GADTs. This has the potential to greatly impact the way programmers program with, and understand, GADTs.

One of the required modifications is that we must work with Kan extensions of functors whose domain categories are `|*|` rather than `*`. This in turn requires, for example, that we use the "equality GADT" `Eql`, rather than function types, in the Haskell implementation of left Kan extensions, and so establishes that a arbitrary GADT can be regarded as mere syntactic sugar for a Haskell data type involving an ordinary existential type and the `Eql` GADT. This means that supporting arbitrary GADTs boils down to supporting just (existential types and) the `Eql` GADT. Of course, programmers may still prefer to program with non-`Eql` GADTs for syntactic convenience, so we give our initial algebra packages relative to both syntactic representations.

The next phase of this research strand is concerned with developing similar results for dependent types. They key to our results for nested types and GADTs is the observation that, although they are not typically presented this way, these types can all be seen as inductive families over appropriate index kinds — `*` for nested types and `|*|` for GADTs. The algebra of reindexing used to accomplish the required change in perspective for nested types and GADTs is represented by functor composition and its left and right adjoints, which are none other than left and right Kan extensions, respectively. Dependent types are also inductive families, but these families are indexed by types rather than by kinds. Nevertheless, the algebra of reindexing required to derive initial algebra packages for dependent types should still comprise composition and its adjoints. In the dependent type setting, composition has type `(I' -> I) -> (I -> *) -> I' -> *`, where `I` and `I'` are index types. If we write $\Delta_{\mathtt{f}}$ for the operation mapping `F :: I -> *` to `F.f :: I' -> *` for `f :: I' -> I`, then its left and right adjoints $\Sigma_{\mathtt{f}}$ and $\Pi_{\mathtt{f}}$ of $\Delta_{\mathtt{f}}$ are the dependent type theory analogues of left and right Kan extensions. We intend to use this observation to derive analogues of initial algebra packages for dependent types.

Another direction for future research involves extending our results for (covariant) GADTs to derive initial algebra semantics and packages for mixed variance GADTs. We expect this can be accomplished by first representing mixed variance GADTs as fixed points of difunctors, and then either applying a 'separation of variables' technique or using Peter Freyd's observation that the fixed point of a difunctor can be represented as the fixed point of an appropriate covariant functor. We intend to pursue this research direction in the not-too-distant future. It should also be possible to extend Pitts' techniques for constructing parametric models of observational equivalence to accommodate inductive and nested types, as well as GADTs, although we expect that this is a significant undertaking and not one we are prepared to embark on just yet. On the other hand, a particularly interesting aspect of our work on nested types and GADTs is its revelation of Kan extensions as enormously valuable for understanding and programming with advanced data types. Indeed, we do not believe it is possible to obtain our results for nested types and GADTs without the shift in perspective Kan extensions afford. Because we feel strongly that Kan extensions deserve to be more widely known in the functional programming community, we have submitted a small journal paper entitled "How I Learned to Stop Worrying and Love Kan Extensions," explaining these categorical constructs from a functional programming perspective, showing how they arise naturally in functional programming, and demonstrating that they allow us to have our expressive advanced data type cake and eat it in a structured and effective way, too.

- *Parametricity and Strong Dinaturality*

  The categorical semantics advanced by the line of work just discussed reduces correctness of `fold`/`build` rules and their generalizations to the problem of constructing parametric models for functional languages which validate initial algebra packages. We thus see that work as prescribing properties that any eventual categorical semantics of functional languages with advanced data types should satisfy. Although Gordon Plotkin and Martín Abadi have shown that every polymorphic function in a calculus which admits a parametric model can be interpreted as a dinatural transformation, this is not enough to validate the categorical properties needed to provide initial algebra packages for advanced data types. In particular, the derivation of `build` combinators that their polymorphic arguments can actually be interpreted as *strong* dinatural transformations. Work in progress with Ghani, Uustalu, and Vene thus aims to prove a stronger result than that of Plotkin and Abadi, namely that polymorphic functions in calculi which admit parametric models can be interpreted as strong dinatural transformations. While this will not be the case for all polymorphic functions, we believe we have identified a large and useful class of functions for which it actually is the case.

4. One More Paper

I have written one additional paper which does not fit into either of the two major strands of my research. "Staged Notational Definitions," written together with Walid Taha, extends MacroML — a language which supports inlining, parametric macros, recursive macros, and macros that define new binding constructs, and which avoids some technical difficulties usually associated with macro systems, such as hygiene and scoping issues, by interpreting macros as multistage computations — to support formal reasoning principles for staged macros. Although MacroML is suitable for *expressing* staged macros, it is not particularly good for *reasoning* about them. In particular, MacroML does not support α-conversion, and so is not only sensitive to renaming of (what appear locally to be) bound variables, but also requires a non-standard notion of substitution. By contrast, SND, our calculus of formal reasoning, does support α-conversion, even while retaining MacroML's phase distinction. The result is a language which is better, in that the semantics of programs does not change if the programmer accidentally "renames" what looks locally like a bound variable. This is achieved using a novel notion of signature which captures precisely how to correctly pass parameters to macros without reintroducing complexity into the process of variable renaming. The semantics of SND is defined by interpreting it into a multi-stage calculus, and our main result shows that soundness of SND's equational theory can be established directly from the equational properties of this calculus.

References

[1] Software Errors Cost the U.S. Economy $59.5 Billion Annually. At `http://www.nist.gov/public\_affairs/releases/n02-10.htm`.

[2] A History of Haskell: Being lazy with class. Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. At `http://research.microsoft.com/~simonpj/papers/history-of-haskell/index.htm`.

[3] A short cut to deforestation. Andrew Gill, John Launchbury, and Simon Peyton Jones. International Conference on Functional Programming and Computer Architecture, pp. 223 – 232, 1993.