

In my research, I aim to advance the state of the art in the design, semantics, verification, and implementation of *strongly-typed programming languages*, with a particular emphasis on language mechanisms for modularity and data abstraction.

Strong type systems have become the norm in mainstream, general-purpose, high-level programming languages such as Java and C#. They provide static *type safety* guarantees, ensuring that programs written in these languages do not suffer from a large class of run-time errors, as well as *data abstraction* guarantees, enabling programmers to write more robust, modular, and reusable code.

However, while strong type systems are critical in making large-scale software development feasible, the type systems of today's languages have become alarmingly complicated. In particular, due to the desire for backward compatibility with legacy code, mainstream languages have evolved primarily through a steady accretion of features. Although this process has provided a pathway for essential concepts like *parametric polymorphism* (“generics”) and *higher-order functions* (“closures”) to break out from their academic origins into real-world programming, the accumulation of multiple programming paradigms (object-oriented, concurrent, functional, imperative) has exploded the complexity budgets of mainstream languages to the point that writing good code in these languages—and reasoning about its correctness—is extremely difficult.

Thus, two key challenges in programming languages research are: (1) designing next-generation programming languages that synthesize the complementary benefits of existing paradigms more harmoniously, and (2) developing viable methods for formal verification of multi-paradigm programs. My research tackles both of these challenges.

As the name of my research group at MPI-SWS suggests, I take as my starting point the *functional* programming paradigm. Functional programming emphasizes the ability to treat functions as a form of first-class data, as well as the ability to isolate and encapsulate *impure* (i.e., side-effecting) computations. I begin with functional programming for several reasons. First, it is closely connected to the foundations of type systems—namely, constructive logic and the λ -calculus—and thus research on extensions of λ -calculus models may be transferred directly to extensions of real functional languages. Second, the functional paradigm encourages *orthogonality* in language design, which in turn makes it easier to understand and develop different language features independently of one another. Lastly, in light of the renewed multicore-driven interest in concurrent and parallel programming, the functional programming paradigm offers a tangible advantage over other paradigms, due to its emphasis on persistent data and effect encapsulation. In summary, functional languages have been at the forefront of foundational programming language research, and I believe they will continue to have a profound impact on the design of mainstream languages that most programmers use.

Using the functional paradigm as my “home base”, I have explored a number of interesting and important problems under the general category of multi-paradigm language design and semantics. Specifically, since one of the primary functions of strong type systems is to support *modularity and data abstraction*, I have focused on studying (1) how to *design* new modularity mechanisms that reconcile and synthesize existing mechanisms from different programming paradigms, and (2) how to *reason formally* about the benefits of data abstraction in multi-paradigm languages. In the remainder of this statement, I will outline the research contributions I have made thus far, as well as several directions for future work.

1 Designing Modularity Mechanisms for Next-Generation Languages

My language design work has centered around the ML module system [19, 20], the most evolved mechanism for data abstraction in the realm of functional programming. ML modules support nested namespace management through *structures*, a rich interface language of *signatures*, client-side abstraction (*i.e.*, generics) through *functors*, and implementor-side abstraction through *sealing*. As a result of its flexibility and expressive power, the module system is one of ML’s most distinctive and widely used features.

While the ML module system serves as an excellent starting point for designing next-generation module systems, it is far from perfect. There are a variety of desirable features offered by other modularity mechanisms (such as Haskell’s *type classes* and DrScheme’s *units*) that ML modules lack. To make matters worse, a number of different formalisms have been proposed for defining the semantics of ML-style module systems, leading to the common perception that the semantics of modules is more complex than it actually is. Over the course of several papers [13, 7, 14, 12, 9, 10, 16, 29], I have tried to combat both of these problems by: (1) showing how to blend ML’s notion of module together with promising ideas from other languages, and (2) developing simple, unified accounts of ML module semantics that are accessible to a general PL audience. In this section, I will describe some of my key results in this area.

Modular Type Classes It has often been remarked that the basic mechanisms of the ML module system are very similar to the basic mechanisms of Haskell’s *type class* system [32], yet they serve quite different purposes—the former supports purely *explicit* control over scope and linking, while the latter supports purely *implicit* program construction and overloading (so-called *ad hoc* polymorphism).

In joint work with Bob Harper and Manuel Chakravarty, I showed how to combine the benefits of ML modules and Haskell type classes in one language [14]. The basic idea is to encode Haskell’s *class* and *instance* declarations directly in terms of existing ML constructs (*i.e.*, signatures, structures, and functors). The implicit composition of these module-level constructs—corresponding to the implicit construction of Haskell’s “dictionaries”—is then supported by a simple extension to ML’s type inference system. This approach, dubbed *modular type classes*, facilitates a clean synthesis of the ML and Haskell paradigms for modular programming without incurring any redundancy of mechanism.

Recursive Modules One of the most glaring limitations of ML modules is that they cannot be defined recursively, thus inhibiting the modular decomposition of mutually recursive type and function definitions into separately compilable components. Consequently, *recursive modules* have been one of the most frequently requested extensions to ML.

Not surprisingly, extending ML with recursive modules is highly non-trivial. The most serious problem that arises in designing such an extension is what in my thesis I termed *the double vision problem* [8]. The problem is essentially as follows: In ML, due to the presence of nested modules, there may be multiple names for (or “paths” to) an abstract type at different points in a program, but at any *particular* point in the program only one such path is in scope. In the presence of recursive module definitions, however, there may be multiple pathnames to the same abstract type in scope at one time, and it is important that these paths all be considered interchangeable by the typechecker. Designing a typechecker that satisfies this property turns out to be incredibly tricky, primarily since the knowledge about the identity of an abstract type may differ in different scopes.

Eventually, I realized that the reason why recursive modules are a “hard” extension is quite fundamental: if we forget ML modules and just try encoding the functionality of *recursive ADTs* in (a simple extension of) the core polymorphic λ -calculus, *System F*, we find that it is not expressible even at that foundational

level. So, I determined that the key to developing an effective cure for the double vision problem was to first show how to solve it at the level of System F, and only then to scale the solution to the level of ML modules. I proposed a core calculus called *RTG*, which extends System F with the essential piece needed in order to define recursive ADTs, namely the ability to *forward-declare* an abstract type before it is defined [9]. I subsequently showed how to use RTG as the foundation of a double-vision-avoiding recursive module extension to ML [10].

MixML The aforementioned work on extending ML with recursive modules left one key problem open: how to support *separate compilation*, or even separate typechecking, of mutually recursive modules. ML’s traditional mechanism for supporting separate compilation—namely, the *functor*—does not scale well to the setting of recursive modules. The body of a functor (which defines its *exports*) may depend on its argument (which specifies its *imports*), but not vice versa. In the context of recursive modules, however, the import specifications of a separately-compiled module may in general need to refer recursively to abstract type components provided in its exports, so functors do not suffice.

To solve the separate compilation problem, Andreas Rossberg (my postdoc at MPI-SWS) and I turned to Gilad Bracha’s concept of *mixin modules* [6]. Originally developed in an object-oriented setting, mixin modules are essentially records with some of the components missing. When two mixin modules are linked, the exports (the defined components) of each module are used to fill in the imports (the missing components) of the other. Thus, mixin modules make recursive linking easy. On the other hand, mixin module systems have traditionally lacked support for type abstraction, an essential feature of ML modules.

To combine the benefits of both ML and mixin modules, we developed *MixML*, a novel module system design that supports recursive linking of separately-compiled modules as a primitive notion, as well as the full expressive power of ML-style type abstraction [16]. We are not the first to propose such a combination—the same basic idea is at the core, for instance, of both Flatt *et al.*’s *units* [25] and the class system in Scala [24]. What distinguishes MixML from its predecessors is that (1) it comprises a small set of very simple, orthogonal mechanisms, and (2) those mechanisms can be combined in various ways to encode *all* the features of the ML module system. In this way, MixML not only *generalizes* the ML module system, it also *simplifies* it.

Modules for Dummies When I began my work on the ML module system, a number of different formalisms had already been developed for it, each with subtly different semantic behavior. In collaboration with Karl Cray and Bob Harper, I defined a unifying type system for modules, under which different ML dialects could be understood as subsystems that pick and choose different features [13]. In my PhD thesis [8], I presented a simpler, cleaner version of this type system, which was subsequently used as the basis for the first mechanized formalization of the metatheory of Standard ML [18].

Most module type systems, including the one from my thesis, employ a weak form of dependent types, as well as a number of other advanced features (such as singleton kinds or translucent sums), giving the impression to many that ML modules require sophisticated type theory. In recent work with Andreas Rossberg and Claudio Russo, I have demonstrated that this is not the case [29]. Although ML modules superficially appear to be dependently-typed, we can give them a dead-simple, abstraction-preserving *elaboration semantics* by directly translating them into plain old System F. We call this the *F-ing modules* approach, and we have already received a lot of positive feedback indicating that our approach makes ML modules much easier for non-specialists—and programmers!—to understand.

2 Reasoning About Data Abstraction in Multi-Paradigm Languages

The central goal of module system design is to provide a means of breaking code into reusable components and imposing data abstraction boundaries between those components. But how do we know that the data abstraction facility provided by a language’s module system actually “works”?

The canonical account of what it means for a language’s data abstraction mechanism to “work” is given by John Mitchell’s notion of *representation independence* [21], which in turn is derived from John Reynolds’ notion of *relational parametricity* [27]. Representation independence means essentially that, given two different implementations of an ADT, if there exists *some* relation between their internal data representations that is preserved by the operations of the ADT, then no client can distinguish between them—*i.e.*, they are *observationally equivalent*. If a language guarantees representation independence, it effectively ensures that the implementation of an ADT is held abstract from clients.

Techniques for proving representation independence were originally developed in the context of (variants of) System F, and over the past two decades there has been a great deal of work on extending them to the setting of more realistic languages. There has not, however, been any prior work on proving representation independence for multi-paradigm languages that support both the functional style of data abstraction, via *universal and existential types*, and the imperative/OO style of data abstraction, via *local mutable state* (local variables or private fields). Handling such a combination is critical for reasoning about next-generation programming languages, since this combination of paradigms is present even in ML, a 20-year-old language!

I have therefore set out to develop effective techniques for reasoning about representation independence (and, more generally, *observational equivalence*) in multi-paradigm languages. This has led to a number of interesting results.

Reasoning About Stateful ADTs In joint work with Amal Ahmed and Andreas Rossberg, I presented the first method for proving representation independence in an ML-like language supporting *both* existential types and general (higher-order) mutable references [1]. Our focus was on reasoning about *stateful* ADTs—that is, ADTs in which the type representation of an existential package is dependent on some piece of local state. (The canonical example of such a stateful ADT is a symbol table module, which exports a type of symbols that represent the valid indices into some private, dynamically generated/updated hash table.)

One key idea in this work is that just establishing *invariants* on local state (as previous methods had allowed) is often not enough—it is useful to be able to establish properties about pieces of local state that may *evolve* over time in a controlled fashion. In addition, we showed that the well-known difficulties with reasoning about higher-order state can be handled easily using *step-indexed logical relations* [3, 2]. Although our primary focus was on proving representation independence for ADTs that exhibit an interaction of existentials and state, our method also handles several well-known simply-typed examples from the literature on local state that have proven difficult for previous logical-relations methods to handle.

Non-Parametric Parametricity Traditionally, the parametric nature of polymorphism and type abstraction is guaranteed statically by a language’s type system. However, some modern programming languages include a useful feature that appears to be in direct conflict with parametric polymorphism, namely the ability to perform *intensional type analysis* [17], *i.e.*, run-time inspection of the identity of an abstract type. By its very nature, type analysis is *non-parametric*, and thus violates the standard parametricity and representation independence properties that hold in its absence.

In order to reconcile type abstraction and type analysis, several researchers have proposed the use of *dynamic type generation* [28, 31]—that is, when one defines an abstract type, one should also be able to

generate at run time a “fresh” type name, which may be used as a dynamic representative of the abstract type for purposes of type analysis. The idea is that the freshness of name generation will ensure that user-defined abstract types are viewed dynamically in the same way that they are viewed statically—as distinct from all other types. However, no one had actually proven that dynamic type generation actually *works*—*i.e.*, that it re-establishes the same parametricity guarantees that abstract types enjoy in traditional languages.

In collaboration with Georg Neis (my student) and Andreas Rossberg, I proved that dynamic type generation does indeed work [23]. As in our work on stateful ADTs, we used a form of step-indexed logical relation, this time in order to account for the combination of type abstraction and stateful type name generation. Furthermore, we showed how in languages with non-parametric operations (like intensional type analysis), it can be useful to distinguish between *positive* and *negative* notions of parametricity, corresponding to whether a module behaves parametrically with respect to its enclosing program context or vice versa.

Logical Step-Indexed Logical Relations The work described above depended critically on the use of a *step-indexed logical relation*. In such a model, program equivalence is defined by induction on the number of steps of computation. While this “step index” provides a handy induction metric, we found that it also clutters proofs of representation independence results with step-index arithmetic, to the point that the main substance of the proofs becomes obscured. Thus, it seems clear that widespread acceptance of step-indexed logical relations will hinge on the development of abstract proof principles for reasoning about them.

The key difficulty in developing such abstract proof principles is that, in order to reason about two programs being *infinitely* logically related—*i.e.*, belonging to a step-indexed logical relation for *any* number of steps, which is ultimately what one really cares about—one must pick an arbitrary n and reason about whether the programs are logically related for n steps, thus forcing one into finite, step-specific reasoning.

In joint work with Georg Neis, Andreas Rossberg, Amal Ahmed, and Lars Birkedal, I presented a solution to this dilemma in the form of two relational modal logics: LSLR [11] (for a purely functional extension of System F with recursive types), and LADR [15] (for an impure ML-like language). Our work here synthesizes ideas from several earlier logics—including Plotkin and Abadi’s logic for relational parametricity [26], Appel *et al.*’s “very modal model” [4], and Yang’s relational separation logic [33]—in a novel way, resulting in the first logic for reasoning syntactically about observational equivalence in a multi-paradigm language with type abstraction and general references.

3 Future Work

There are several directions for future work that I would like to pursue.

In terms of future design work, the most pressing question that I am interested in investigating is how to cleanly synthesize the design of MixML with my earlier work on modular type classes. On a related note, I am interested in exploring whether the “F-ing modules” approach (described at the end of Section 1) can be used to simplify the formal specification of modular type classes. Lastly, given my success in fusing mixin composition (originally an object-oriented concept) with ML modules, I am also keen to explore further how other good ideas from the OO world can be effectively incorporated into a functional-language setting.

In terms of semantics, one important problem is to generalize our proof methods in order to handle a language with concurrency. Another problem is that, although many techniques have been developed for relational reasoning about programs, there is still not a clear understanding of how the presence or absence of different language features (such as general references or control operators) affects the principles for reasoning about program equivalence. In collaboration with Georg Neis and Lars Birkedal, I am in the midst of preparing a paper on this topic, in which—taking inspiration from game semantics—we propose a unified

framework of (step-indexed) logical relations for languages with different combinations of effects. Finally, I would like to investigate the connections between our logical-relations methods and related approaches for reasoning about multi-paradigm languages, such as Sumii *et al.*'s *environmental bisimulations* [30] and Nanevski *et al.*'s *Hoare type theory* [22].

Although my work so far has focused primarily on the foundations of next-generation language design and semantics, I believe it is also important to build actual implementations of my language design proposals, and to apply my proof methods to the mechanical verification of realistic programs. My collaborators and I have already made some initial steps in both these directions. On the implementation side, we have begun to build a serious implementation of MixML (based on the Moscow ML compiler) which we plan to use as a testbed for our multi-paradigm language design ideas. On the verification side, we have begun to mechanize our step-indexed logical relations in the Coq proof assistant. Thanks to the POPLmark challenge [5], mechanization of programming language metatheory has become a hot topic in recent years, but most of the effort so far has produced techniques for easing the mechanization of *type safety* proofs. In contrast, the soundness proofs for advanced semantic models (such as our step-indexed logical relations) are significantly more complex and easier to get wrong. Mechanizing such proofs is thus, we believe, an important undertaking, as well as a challenging research problem in its own right.

References

- [1] Amal Ahmed, **Derek Dreyer**, and Andreas Rossberg. State-dependent representation independence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.
- [2] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.
- [3] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.
- [4] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.
- [5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: the POPLmark challenge. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs)*, 2005.
- [6] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *International Conference on Computer Languages (ICCL)*, 1992.
- [7] **Derek Dreyer**. A type system for well-founded recursion. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.
- [8] **Derek Dreyer**. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005.
- [9] **Derek Dreyer**. Recursive type generativity. *Journal of Functional Programming*, 17(4&5):433–471, 2007. An earlier version appeared in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2005.
- [10] **Derek Dreyer**. A type system for recursive modules. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2007.
- [11] **Derek Dreyer**, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2009.

- [12] **Derek Dreyer** and Matthias Blume. Principal type schemes for modular programs. In *European Symposium on Programming (ESOP)*, 2007.
- [13] **Derek Dreyer**, Karl Cray, and Robert Harper. A type system for higher-order modules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003.
- [14] **Derek Dreyer**, Robert Harper, and Manuel M. T. Chakravarty. Modular type classes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.
- [15] **Derek Dreyer**, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful ADTs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2010.
- [16] **Derek Dreyer** and Andreas Rossberg. Mixin' up the ML module system. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.
- [17] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1995.
- [18] Daniel K. Lee, Karl Cray, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.
- [19] David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, 1984.
- [20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [21] John C. Mitchell. Representation independence and data abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1986.
- [22] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5&6):865–911, September 2008.
- [23] Georg Neis, **Derek Dreyer**, and Andreas Rossberg. Non-parametric parametricity. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2009.
- [24] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2005.
- [25] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2006.
- [26] Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, 1993.
- [27] John C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, 1983.
- [28] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2003.
- [29] Andreas Rossberg, Claudio V. Russo, and **Derek Dreyer**. F-ing modules. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2010.
- [30] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2007.
- [31] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2005.
- [32] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1989.
- [33] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1–3):308–334, 2007.