Research Statement
Matthew Fluet

My main research interests lie with the design and implementation of programming languages, including type systems, program semantics, and compiler technology. More specifically, I am interested in the ways in which expressive programming languages and their type systems may facilitate the development and maintenance of programs both for correctness and for efficiency. Both of these facets are of great importance to the development of robust software. Static typing has proven very effective at ruling out erroneous behaviors and at catching bugs early in the software development cycle. However, the type systems of today continue to have limitations, failing to admit the expression of some correctness properties of programs and failing to admit the expression of low-level, resource and performance conscious programs. Research in type systems, in particular, and programming languages, in general, should be pursuing advances along both of these fronts. I believe that the best research in this field emerges from a healthy balance of formal theory and practical implementation. When practiced in this manner, research in programming languages can be an agent of “technology transfer,” bringing formal results to justify practical techniques.

I have applied this philosophy to my own research by first noting compelling applications of interesting language features and then exploring how to better explain and refine these features, particularly their supporting type systems. As one example, my thesis work focuses on applications of and type systems for the region-based memory management features of Cyclone, a type-safe dialect of C. I implemented, in Cyclone, a simple copying garbage collector for a Scheme interpreter [SPACE’04]. This implementation relied crucially upon advanced features of Cyclone in order to ensure the type safety of the garbage collector. Since a formal model that justified these advanced features was lacking, I pursued a line of research that offers a simpler account of these features, whereby it is possible to encode the key features of Cyclone in a uniform target language, for which type safety may more easily be established [ICFP’04, TLCA’05, ICFP’05, ESOP’06].

My collaboration with Riccardo Pucella to explore the limits of the phantom-types technique [TCS’02] is another example. This programming technique uses parametric polymorphism, type constraints, and unification of polymorphic types to enforce program invariants in languages with type systems that, upon first inspection, appear to be too weak for this purpose. Others had demonstrated the practicality and the versatility of this technique for embedding domain specific languages. Our contribution was to formally justify this technique by designing an expressive source language that captures programs expressible by the technique. This formal investigation, in turn, yielded a novel application in the form of datatype specializations [MLWRK’05].

On the more practical side, I continue to collaborate with Stephen Weeks and Suresh Jagannathan on the development of MLton, a whole-program optimizing compiler for Standard ML, including the development of novel optimizations [ICFP’01]. There are a number of theoretical and practical questions that remain to be explored in the context of MLton, many of which could be cast as undergraduate or masters student projects. I am particularly interested in understanding how MLton’s compilation model can be extended to richer input languages without sacrificing
Balancing formal theory and practical implementation will continue to be a principle I practice in future research endeavors. The memory management scenarios discussed below are one instance of a more general phenomenon that is not captured well by conventional languages and type systems: type states or statically verified program-point specific properties and relations. Such notions are useful for handling many sorts of resources, including abstract resources representing proofs of properties or capabilities to perform some action. A long-term research goal is to investigate these sorts of systems, considering both their application to new domains and the design considerations that would make them palatable in a high-level programming language.

I am also particularly interested in forming research projects that bring the strengths of programming languages research to other domains. As a sub-discipline, programming languages addresses questions about how to correctly and efficiently describe computations. Understanding what computations other computer science researchers wish to express, we can work towards advances in how those computations may be best expressed.

Region-based Memory Management

My thesis work explores type-systems for region-based memory management. Region-based memory management offers a healthy compromise in tuning a program’s memory usage; by reclaiming groups of related objects wholesale, region-based languages may avoid both the inefficiency of garbage collectors and the tedium of malloc/free. Type systems for region-based languages further enhance their utility by statically catching (possibly) erroneous use of the region primitives. In my dissertation, I examine three “flavors” of type systems for region-based languages:

- a type-and-effect system (a la Tofte-Talpin)
- a monadic type system
- a substructural type system

I demonstrate that one may successively encode the type-and-effect system into the monadic type system and monadic type system into the substructural type system. I discuss these in more detail below, but the key insights are as follows. In the first translation, we are able to trade the subtleties of an effect system for the simplicity of a monadic system, requiring only parametric polymorphism (a la System F) for encapsulation. In the second translation, we are able to relax the requirement that regions have last-in-first-out (LIFO) lifetimes following the block structure of the program, yielding a more expressive language.

While it is tempting to infer that one can order these three “flavors” of type systems by increasing expressiveness, in general, the relationships between them are much more complex. A more accurate picture is given by a Venn diagram, where region-based memory management falls into the intersection of these three “flavors” of type systems, as one feature that may be handled by each of them. I plan to explore how these different type systems (and others) relate to one another and to various features of programming languages. The goal of this research is to be able to discern “the right tool for the right job.” I believe that this will yield benefits both in external programming languages, where language features are matched with complementary type systems that facilitate reasoning about correctness, and in internal compiler intermediate languages, where low-level run-time behavior may be accurately analyzed and safely transformed.
Applications

Nonetheless, having “the right tool for the right job” is not an end in and of itself. As I’ve discussed above, my research has been guided by first understanding the needs of practical applications, and then exploring how to better explain and refine the language features that support these applications.

One major application which has guided my research is the Cyclone project. Cyclone is a type-safe dialect of C that is intended to give programmers as much control over memory management as possible while retaining strong, static typing. Cyclone’s initial design was based upon a simple, lexically-scoped region system, but has since been extended with a number of novel features (dynamic regions, unique pointers, etc.). The efficacy of these new memory management features has been detailed, including their use in a streaming media server and a space-conscious web server. I consider one other application in more detail.

In collaboration with Dan Wang, I implemented, in Cyclone, a simple Scheme interpreter and copying garbage collector that manages the memory allocated by the interpreter [SPACE’04]. Since our interpreter, as a Cyclone program, had no garbage collector, our implemented garbage collector could not rely upon a host collector to reclaim the from space at the end of a copying collection. Hence, all of our memory management was done manually, yet type safely. Our preliminary benchmarks demonstrated that one can indeed build a system with reasonable performance when compared to other approaches that guarantee safety. More importantly, we could significantly reduce the amount of un(type-)safe code needed to implement the system. This implementation relied crucially upon both dynamic regions and unique pointers in order give a sound type to the forwarding pointer of Scheme objects.

While the soundness of Cyclone’s lexically-scoped regions had been established, a model that justified the soundness of the new features was lacking, due to sheer complexity. Hence, having both compelling applications of the features and a need to better understand their properties motivated the study of simpler models and type systems, where it would be possible to encode the key features of Cyclone in a uniform target language.

Substructural Type Systems

Advanced type systems for state rely upon limiting the ordering and number of uses of data and operations to ensure that state is handled in a safe manner. For example, (safely) deallocating a data structure requires that the data structure is never used in the future. In order to establish this property, a type system may ensure that the data structure is used at most once; after one use, the data structure may be safely deallocated, since there can be no further uses. As another example, a resource conscious application may require that each acquired resource is put to use by the program, in order to avoid wasting resources. To establish this property, a type system may ensure that a resource is used at least once.

A substructural type system provides the core mechanisms necessary to restrict the number and order of uses of data and operations. In collaboration with Amal Ahmed and Greg Morrisett, I have explored a number of novel uses of substructural type systems applied to the problem of handling mutable state in programming languages [TLCA’05, ICFP’05, ESOP’06]. We have demonstrated that a substructural type system can provide foundational typing support for: deallocation of references; strong (type-varying) updates; storage of unique objects in shared references; and region-
based memory management.

There are a number of open questions that remain in this area. Of particular importance is exploring language features that help manage the complexity of these type systems. In the core calculi considered above, administrative details can become burdensome to a programmer. The investigation of inference algorithms and flow analyses should yield insights that make these systems more convenient in source languages.

**Type-and-Effect Systems**

Type-and-effect systems developed from a line of research aiming to combine the advantages of functional and imperative programming. Just as a type describes what an expression computes, an effect describes how an expression computes. In particular, the effect of an expression serves as a concise summary of the observable side-effects that the expression may generate when it is evaluated. Variation amongst type-and-effect systems largely arises from the choice of effect language and the choice of auxiliary judgments that prove when one effect term is equivalent or subsumed by another effect term. Virtually every type-and-effect system includes a number of atomic effects (e.g., read mutable state, write mutable state, throw exception $E$) and an algebraic structure for combining effects; the most common algebraic structure is simply sets of atomic effects.

I see a growing need to better understand the range of type-and-effect systems. Effect terms as sets of atomic effects have been studied, but have limited application. At one end of the spectrum are applications where it suffices to distinguish between the presence or absence of an effect (e.g., I/O interaction, non-termination, dynamic behavior). For example, expressive type systems often introduce “administrative” terms that are present for typing purposes only, and have no run-time significance. An important open question I am interested in pursuing is whether we can exploit the simplicity of these effect terms to yield simpler type-and-effect systems.

At the other end of the spectrum are applications where it is necessary for effect terms to accurately reflect run-time behavior of a program. For example, compilers are often conservative in the presence of effects like exceptions, because transformations that change the order of observed exceptions are not semantics preserving. Richer effect algebras offer a means by which program transformations may be enabled or disabled based on whether or not they preserve the effect of an expression. An important direction for my future research is to hone in on suitable effect algebras that offer the right trade-offs between expressiveness and tractability.