The abstraction that a programming language provides influences the structure and algorithmic complexity of the resulting programs: just imagine creating an artificial intelligence engine using assembly or building an operating system in a language that does not allow direct access to memory. I want to use programming language design principles to improve the reliability, security, maintainability, and extensibility of software systems. In particular, I am interested in the development of type systems that provide good abstractions for modular, reusable code.

The ubiquity of research concepts in mainstream production code demonstrates the importance of programming languages research. The upgrades made to C by C++, one of the most widely used production languages, were based on object-oriented principles from Simula and on generic programming features in Ada[1]. And let us not forget Lisp, the language based on McCarthy's adaptation of Church's lambda calculus. Who would have expected such a simple and elegant computation model to be useful enough to power commercially successful companies such as Graham's Viaweb and the airline pricing company ITA? The lambda calculus has inspired advances even in systems development: Google's MapReduce paradigm is based on Lisp's list processing functions map and reduce. MapReduce, used for parallelizing processes across arbitrarily large numbers of machines, has greatly increasing Google's effectiveness in harnessing computing power[2].

Because those like McCarthy and Stroustrup have set the groundwork for high-level language design, language designers can now concern ourselves with how to enlist the language's type system in ensuring code correctness and security. This brings us to type systems, the future of programming languages research. Type systems provide useful guarantees often crucial to reasoning about program correctness. A type safe language will guarantee that any well-typed program will terminate with a correct answer or raise an exception. A type system protects against perils ranging from using the wrong arguments to division-by-zero errors. The newest popular languages (Java, ML, Haskell) are all type safe: exporting this correctness-checking to the type checker relieves the programmer of unnecessary responsibility.

I am interested in using run-time type analysis to provide more expressive language constructs for polymorphism, the concept of allowing a single function to be defined on multiple types. There are two forms of polymorphism: parametric, which executes the same code regardless of argument type, allowing the programmer to define a single function for multiple types, and overloading, or ad-hoc polymorphism, which examines the type of the argument to determine which function to execute, allowing the programmer to define data type-specific behaviors. Polymorphism is difficult to implement because the compiler must choose the correct version of the operator, inserting transfer (casting) functions where appropriate. Language support for polymorphism is useful: one of C++'s major advantages over C is its generic template metaprogramming features.

I have been working with Professor Greg Morrisett on type systems and polymorphism in

1 Stroustrup, B. 2007. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California, June 09-10, 2007). HOPL III. ACM Press, New York, NY, 4-1-4-59.

2 Dean, J . and Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium of Operating Systems Design & Implementation – Volume 6* (San Francisco, CA, December 06-08, 2004). UESNIX Association, Berkeley, CA, 10-10.

Haskell, a strongly typed functional language using the parametrically polymorphic Hindley/Milner type inference algorithm. It has support for ad-hoc polymorphism with its type classes, which were developed as a generalization to the Hindley/Milner system to allow for the overloading of functions such as equality and arithmetic operators[3].

Here I must make the distinction between static and dynamic type-checking. Type classes are *statically checked*, meaning all types and associated function dispatches are determined by the compiler. While static checking provides certain guarantees before runtime, it can be limiting because there exist types that cannot be determined at compile time. For instance, it is impossible for a program such as an interpreter to anticipate the type of the result of each input it processes because the types depend on user input.

Fortunately, Haskell also provides a way of performing *intensional type analysis*, or examining the types at run time. In 2002 Baars and Swierstra introduced dynamic typing concepts that were eventually incorporated into Haskell's `Dynamic` and `Typeable` libraries[4]. `Typeable` provides us with with the *type-safe cast* operator, which can return whether an object has a given type.

Lammel and Peyton Jones have shown that armed with type classes and type-safe cast, the Haskell programmer can use *generic programming* methods[5], which allow the programmer to write algorithms that can be executed on data structures of different types using the appropriate traversal mechanisms. A weakness of this original approach is that it requires all types to be supplied at once, excluding the possibility of using these generic methods with type classes. Lammel and Peyton Jones recognized this and proposed abstracting over type classes to solve this problem[6]. There are various issues with this solution: it is unsound, the abstraction forces the programmer to anticipate class parameters, and it requires a modification of standard classes with the additional parameter. Another issue is that this approach is also static.

My goal is to find a solution to the problem of extensible type classes that allows us to take advantage of type classes dynamically. Using dynamic type analysis would allows more flexibility when examining types and allow for the support of of dynamic types. This solution would be useful not only in any Haskell program using dynamic types and type classes, but this mechanism would also be quite useful in programs such as embedded languages, for which dynamic types are necessary, and in embedded interpreters, which are quite useful . Adding this extension would benefit not only Haskell but other languages supporting generics. Though Haskell is not widely used in production code, features of other languages such as C++ and Java are often based on concepts developed in research. There has been much comparison of C++ templates with Haskell type classes, and there has been talk that C++ is coming out with a new addition that have been said to "smell like Haskell type classes."

3 Walder, P. and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60-76. ACM, January 1989.

4 Baars, Arthur I. And S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference onf Functional programming languages*, pages 157-166.

5 Lammel, Ralf and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26-37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

6 Lammel Ralf and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005),* pages 204-215. ACM Press, September 2005.