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 disallows direct access to memory. I want to use programming language design principles to improve the reliability 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 Simula's object-oriented principles and on Ada's generic programming features\(^1\). Lisp, the language based on McCarthy's adaptation of Church's lambda calculus, has proven useful not only for artificial intelligence researchers but also for commercially successful companies such as Viaweb and ITA. The lambda calculus has inspired advances even in systems development: Google's MapReduce paradigm, based on Lisp functions map and reduce, is used for parallelizing processes across arbitrarily large numbers of machines. MapReduce has greatly increased Google's effectiveness in harnessing computing power\(^2\).

Types are the future of programming languages research. While a typed language performs basic checks to ensure type compatibility, a type safe language guarantees that any well-typed program will either terminate with the correct answer or raise an exception. Not only do type systems provide useful guarantees for reasoning about program correctness, but they also protect against perils ranging from wrong arguments to division-by-zero errors. The newest widely-used languages (Java, ML, Haskell) are all type safe: exporting this correctness-checking to the type checker relieves the programmer of unnecessary responsibility.

I have been working with Professor Greg Morrisett on type systems and polymorphism in Haskell, a strongly typed functional language. I am interested in using run-time type analysis to provide more expressive language constructs for polymorphism, which is the concept of allowing a single function to be defined on multiple types. Haskell has support for ad-hoc polymorphism with its type classes, which were developed to allow for the overloading of functions such as equality and arithmetic operators\(^3\). Though useful, type classes are restricted by compile-time checking limitations. I propose to find a way to examine type classes at run time.

Being able to examine type classes at run time would provide the programmer more flexibility in creating polymorphic algorithms. 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. Language support for polymorphism is useful: one of C++'s major advantages over C is its generic template metaprogramming features.

Polymorphism is difficult to implement because it involves choosing the correct version of the operator. Haskell's type classes are statically checked, meaning that the compiler determines all types and associated function dispatches before run time. While static checking provides certain guarantees before run time, 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. Using dynamic, or run time, type analysis would allow more flexibility.

Haskell also provides support for intensional type analysis, or examining types at run time. Haskell's `Typeable` class for dynamic typing provides a type-safe cast operator, which returns whether a given object has a given type. This is very useful: in Scrap your Boilerplate Lammel and Peyton Jones show that type classes and type-safe cast allow the use of generic programming methods, which allow the programmer to write algorithms that can be executed on data structures of different types using the appropriate traversal mechanisms. Generic programming methods are desirable because they increase programmer productivity and reduce code size.

Unfortunately, since the Scrap Your Boilerplate method requires all types to be supplied at once, we cannot use its generic methods with type classes. Lammel and Peyton Jones recognized this and proposed abstracting over type classes to solve this problem. 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.

I am interested in finding a way to dynamically take advantage of type classes. Among other things, this would allow the use generic methods with type classes. Such a mechanism would be 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 has been rumored to “smell like type classes.”

---

