Research Statement

Richard A. Eisenberg

The goal of my research is to increase software reliability by enabling programmers to give precise, machine-checkable specifications for their algorithms. An incorrect function with a precise specification can be rejected by a compiler, avoiding the possibility of misbehavior at runtime. My tool of choice thus far has been dependent types. My dissertation studies what is needed to **add dependent types to Haskell**, a pure functional language with a growing industrial uptake. The research includes **my implementation of dependent types** into Haskell's main compiler; **the release of this feature will mark the first time dependent types have been available in a language used by programmers in industry**. I expect my next steps to include code optimizations enabled through dependent types and improving compiler support for refinement types, a commonly used subset of dependent types.

Dependent types: a powerful feature for program verification and expressive types

Type systems are one of the great practical successes of programming language theory. Theoreticians define the structure of types, typing rules, and an evaluation relation for a given programming language; using these definitions, they can then prove that a well-typed program does not "go wrong". With this work in hand, a compiler writer can then implement a type checker that confirms to programmers that their real, well-typed programs do not "go wrong" — at least not in the way the type system is designed to avoid. The way in which the theoretical work directly influences a practical artifact is the success of programming language theory.

However, standard (non-dependent) type systems go only so far. They encode information about the general nature of terms in the running program—for example, whether a given term represents a number or a string—and not about specific values. The use of *dependent* types builds on decades of successful use of non-dependent types to protect against programmer errors.

Dependent types can depend on the values of terms. For example, suppose that the variable *row* stores the data from a row of a database, and the *getRow* function projects one field out from a *row*. When we say *getRow row* `"name"`, we know the result is a string. When we say *getRow row* `"gpa"`, we know the result is a number. The use of dependent types allows the *type* of *getRow row* `"name"` to depend on the *value* `"name"`. If we swap *getRow row* `"gpa"` for *getRow row* `"name"` in our grade-comparison code, we get a type error at compile time. There is no chance of a runtime error. I have published a worked-out example of this general idea [3, Section 5].

More broadly, dependent types allow programmers to put arbitrary specifications in types. Type-checking guarantees that the implementation meets the specification. For example, the function *sort* can have a type that admits only functions that actually sort. That is, the type says that *sort* takes a list of numbers and returns another list of numbers, in ascending order, that is a permutation of the input list. If the body of *sort* does not implement a proper sorting operation, it is rejected at compile time.

Because they are so expressive, **dependent types can bridge between static typing (e.g., Java or Haskell) and dynamic checks (e.g., Python or JavaScript)**. Programmers sometimes prefer dynamic languages because they can express idioms (such as lists containing elements from a variety of types) that statically-typed languages struggle with. However, for a data structure to be useful, the programmer must have *some* structure in mind for the data. The brand of dependent types I have implemented, which allows functions to appear in types, can express finely detailed data structures, enabling a static type for the structure the programmer desires and avoiding the dynamic checks.

Research Program: adding dependent types to Haskell

Haskell has always had a tradition of strong, static types. It is the world leader in type system innovation, while still retaining an industrial user base. Haskellers generally want the type system to prevent a larger class of errors than is practical in many other languages. For example, Haskell's types distinguish functions that mutate heap values from other functions (called *pure*); trying to mutate a heap value from pure code is a type error. Using informative types like this simplifies the programmer's job by including more about a function's behavior in its interface. Haskellers, thus, are eager for dependent types, which would allow their types to specify yet more behaviors and make programs safer. Descriptions of how to fake limited dependent types in Haskell stretch back over a decade.

My dissertation, *Dependent Types in Haskell: Theory and Practice*, describes the design and theory of dependent types in Haskell. A key component of this work is my implementation of dependent types in the main Haskell compiler—the Glasgow Haskell Compiler (GHC), to which I am a core contributor. The dissertation also includes a proof of type safety and a detailed accounting of my novel type inference algorithm. My work has the support of the Haskell community, and I expect that a future release of GHC will include my implementation. That release will put dependent types in the hands of programmers outside the ivory tower for the first time.

Merging types and kinds The design of dependent types in Haskell is based on the type system described in the paper I presented at ICFP'13 [8], a top conference in the field. A *kind* in Haskell classifies a type, similar to how types classify terms. Previously, kinds and types were treated differently, but incorporating dependent types requires that kinds be as expressive as types. My ICFP'13 paper achieves this by merging the language of kinds and types. I have since implemented this change—a large step toward dependent types that has been highly anticipated within the Haskell community—and it will be included in the next release of GHC.

Type families Another core component of Haskell's dependent types is type-level computation. Haskell uses *type families* to represent functions on types. We can define a type family *F* such that *F Int* is the same type as *Bool*. In the course of developing dependent types, I have studied and refined this feature several times [2, 6], most notably by introducing *closed* type families at POPL'14 [4], another very competitive venue. My implementation is currently used by 89 open-source Haskell libraries which make use of the feature.[1] Closed type families were instrumental in a library I wrote for compile-time dimensional analysis, called *units*. With this library, adding meters to seconds results in a compile-time type error, but adding meters to feet results in an internal unit conversion. Perhaps using such a system would have prevented the Mars Climate Orbiter disaster, among other unit-related mistakes. I collaborated with an astrophysics researcher to write an experience report [5] covering how the library is helpful in the physical sciences.

Faking dependent types In order to guide my research in dependent types—and to give other Haskellers an opportunity to experiment—it is helpful to have a prototype. I thus produced the *singletons* library, presenting a peer-reviewed paper about it at the Haskell Symposium, 2012 [3]. This library uses existing features of Haskell to fake a limited form of dependency, using singleton types. Currently, 47 published libraries depend on *singletons*. The library, through my active maintenance, continues to be a breeding ground for new ideas about dependent types.

[1] Statistic produced by search of libraries from `http://hackage.haskell.org`, Sept. 15, 2015.

Safe coercions Another thread of my research concerns allowing type-safe coercions, performed with no runtime cost, between values known to share a runtime representation. For example, we might say that the type *HTML* is just like *String*, but with extra invariants. Only certain privileged functions that establish the invariants can create values of type *HTML*. Conversion from *HTML* to *String* is always valid. It would thus be convenient to convert, say, a tree of *HTML* values into a tree of *String* values, with no action required at runtime. It is subtle to determine when this operation is type-safe. I implemented GHC's *roles* mechanism (which I presented at ICFP'14 [1], with a detailed proof of type safety) to do exactly this, via a static analysis.

Impact: studying Haskell yields collaborators and makes my work relevant to industry

I have chosen to implement my ideas in GHC to increase the impact of my work. It is possible to design a theoretical system, prove properties about it, build a toy implementation of it, and in so doing make a substantial research contribution. However, such work tends to stay cloistered within academia. In contrast, I have implemented my ideas in an industrial-strength compiler with a sizable user base. Doing so broadens my reach, gives me an active feedback loop, and provides a well-spring of potential collaborators from the Haskell community. In my role as a professor, my access to GHC will also be attractive to students who might relish the opportunity to implement their ideas in the context of a working, real-world compiler.

Haskell seems now to be reaching an inflection point, with increasing industry interest. Haskell teams can be found at Amgen, Bank of America, BAE Systems, Bluespec, S&P Capital IQ, Facebook, Galois, The New York Times, and Standard Chartered, among many others.[2] Research on Haskell has also attracted the interest of the US government; Haskell is used as a core component in a US government program to develop high-assurance software for unmanned aircraft.[3] Because of this growing interest, now is a great time to be working with Haskell, a fact that should draw in both student and new peer collaborators.

Future work

I see several different directions my research can go from this starting point.

Refining dependent types: improving the user experience Once Haskell has the raw power of dependent types, a usability hurdle will remain. Although powerful, dependent types are notoriously hard to work with. A natural next step once the core features are available is to improve their accessibility. An intriguing line of research into refinement types for Haskell [7] includes a much simpler programmer-facing interface than do full dependent types. Refinement types are a weaker subset of dependent types, but refinement-type research suggests that it would be convenient to programmers to have a special syntax for refinement types.

A key step along this path would be to design *first-class, lightweight existential types*. An existential type has some property that is not directly evident in the type itself. As an extreme example, all non-final class types in Java are existential, because the type, say *Object*, does not tell you what type the runtime inhabitant will have—that runtime inhabitant can be of any subtype of *Object*. The refinement types approach to Haskell might state, for example, that the result of a *length* function

[2] `https://wiki.haskell.org/Haskell_in_industry`
[3] `http://smaccmpilot.org/`

is a nonnegative integer. This could be written as the type $\{n : Int \mid n \geqslant 0\}$. In that notation, n is existentially quantified—that is, the type mentions *some* nonnegative integer n, but not a specific one.

Haskell currently supports existentials via a much more verbose syntax, requiring a separate datatype definition for each existential type and manual annotations to pack and unpack the existentially quantified type variable. However, it may be possible to reduce this burden, by devising a scheme where existential types are eagerly unpacked (that is, the existentially-quantified variable becomes available as soon as possible) but lazily packed (that is, the existential type wrapper would be used only when necessary). A naive implementation of this idea in the context of today's Haskell would cause an embarrassing runtime slowdown, however, as the packing and unpacking operations cannot yet be erased. To be truly lightweight, I would need to resolve this thorny issue. With the lightweight existentials, it would be straightforward to use a more natural syntax to express the refinement-type subset of dependent types, encouraging their use more widely.

Optimizing dependent types Statically typed languages have a speed advantage over dynamically checked languages: in a typed language, no runtime type checks are required. Having dependent types in an optimizing compiler opens up the door to studying optimizations of dependently-typed code. Because more about the values is known at compile time, fewer runtime checks may be necessary.

The *getRow* function from the introduction takes a row of data and the name of a field to project out from the row. How is this implemented? Given that *getRow* must work with the field name, an efficient implementation would look up the name in a hashtable. Yet even this efficient lookup takes time. Using dependent types, the *row* argument to *getRow* would have a type that contains the schema for the row (that is, the field name & type information for the columns in the table). This is enough information for us to inline a call to *getRow row* `"name"` to access just the right word in memory, without any comparisons necessary. Or, suppose we do not have the field name directly in the code, but type inference tells us that the result of *getRow row fieldName* is a *String*. If the schema for *row* has only one *String* entry, then we can similarly inline the call, avoiding even looking at the field name.

Human-compiler interaction: letting the programmer get the compiler unstuck Writing a computer program is currently very transactional: the programmer writes code, and then the compiler analyzes the code, accepting it or issuing errors. Although not within the confines of dependent types or Haskell, one of my continuing interests is in improving this interaction. One promising avenue for improvement is by designing an interactive, iterative type inference process.

My idea for interactive type inference works like this: if the compiler process encounters a type ambiguity, it asks the user for help. In the query posed to the user, the compiler is careful to present the user with only those options that would support a well-typed program. The user then chooses the most suitable type for some subexpression and the process continues. If the compilation succeeds, the compiler could either annotate the code directly or produce a supporting file with extra information, used in subsequent compiler runs.

This general approach seems applicable to any language with the possibility of type ambiguity. This set of languages includes Haskell (with ambiguity arising from type classes and also in dependent pattern matches), Java (with ambiguity arising with variable-arity and overloaded functions) and C++ (with ambiguity arising from automatic conversions and overloaded functions), among others. This technique also seems very applicable in the context of novice programmers, as it perhaps can be tuned to fill in—with the programmer's assistance—missing pieces of a program or its types.

With my research experience, I am well poised to continue making impactful research in the theory and practice of dependent types, and I look forward to examining other ways types can serve programmers better. My involvement with Haskell and GHC increases the visibility of my research and provides a platform for my collaborators—students and colleagues alike—to make their contributions known. I am excited to see where the next steps lead.

References

[1] J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[2] R. A. Eisenberg and J. Stolarek. Promoting functions to type families in Haskell. In *Symposium on Haskell*, Haskell '14. ACM, 2014.

[3] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[4] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[5] T. Muranushi and R. A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2014.

[6] J. Stolarek, S. Peyon Jones, and R. A. Eisenberg. Injective type families for Haskell. In *Haskell Symposium*, Haskell '15. ACM, 2015.

[7] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. Refinement types for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[8] S. Weirich, J. Hsu, and R. A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.