Eliminating Bugs with Dependent Haskell (Experience Report)

Noam Zilberstein
Facebook
California, USA
noamz@fb.com

Abstract

Using dependent types in production code is a practical way to eliminate errors. While there are many examples of using dependent Haskell to prove invariants about code, few of these are applied to large scale production systems. Critics claim that dependent types are only useful in toy examples and that they are impractical for use in the real world. This experience report analyzes real world examples where dependent types have enabled us to find and eliminate bugs in production Haskell code.

CCS Concepts: • **Software and its engineering** → **Software development techniques**; **Functional languages**.

Keywords: Haskell, Dependent Types, GADTs

ACM Reference Format:
Noam Zilberstein. 2020. Eliminating Bugs with Dependent Haskell (Experience Report). In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3406088.3409020

1 Introduction

As software systems grow more complicated, it becomes increasingly difficult to reason about end-to-end correctness. It is therefore attractive to use languages with highly expressive type systems in order to prove invariants about code at compile time. These techniques are well known within the Haskell community, but claims that they can be used to increase correctness in practice are largely unsubstantiated. Critics dismiss these techniques as academic exercises that are impractical for use in the real world because they are too complicated and not powerful enough.

Dependent types *are* practical for use in production code. In fact, not only are they practical, but they are effective at eliminating bugs. Through real world examples, we demonstrate that bugs can be discovered and removed at scale using dependent types. These bugs range from accidental coercions and missing inputs to infinite loops.

Haskell is an excellent language to demonstrate dependently typed programming in a real world setting. Recent advancements in the Dependent Haskell [13] project have increased the expressiveness of the Haskell type system, opening up new possibilities for encoding correctness guarantees in code. Haskell is also a performant, natively compiled language that is suitable for use in large scale systems.

At Facebook, Haskell is used to write abuse detection rules as part of a system called Sigma [8]. These rules span a wide variety of abuse types including spam, fake accounts, and fraud. The correctness of this code is crucial; Sigma makes it possible for us to identify and block malicious interactions *before* they affect people on Facebook. Due to the adversarial nature of abuse detection problems, Sigma code must be deployed to production quickly. This further necessitates strong correctness guarantees at compile time. Sigma is also a large scale use-case as it is invoked on nearly every interaction that a user has on Facebook (over one million times per second).

Previous work has shown that dependent types can be successfully deployed in production code [2]. This report builds on top of previous work to show *how* the use of dependent types prevents bugs in practice. In particular, we make the following contributions:

- Section 2 discusses what it means to use dependent types in Haskell and why Haskell's relatively weak form of dependent types presents some benefits.
- Section 3 introduces the Thrift compiler, a motivating use case that we refer to in the rest of the paper.
- Sections 4, 5, and 6 examine different techniques that are used to prove correctness invariants, show what real-world bugs they prevent, and give additional examples for how those techniques can be applied to different types of problems.

2 Dependently Typed Programming in Haskell

The primary goal of programming with dependent types in Haskell is to express invariants about code at the type level.

Haskell '20, August 27, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8050-8/20/08.
https://doi.org/10.1145/3406088.3409020

These invariants are checked by the Haskell compiler (GHC) and therefore any code that compiles must adhere to them. This benefit is seen during initial development as well as during refactoring.

While Haskell's type system is more expressive than that of most mainstream languages, Haskell is not truly a dependently typed language. More concretely, there *is* a phase separation between types and terms. This means that the invariants that can be expressed are limited. However, this limitation also comes with an upside; Haskell has a more powerful type inference system than most dependently typed languages. In practice, Haskell's constraint solver can automate more invariant checking than that of languages like Agda [7].

Although the guarantees are weaker than those of a true dependently typed language, we obtain correctness guarantees that go beyond regular static typing by expressing more information about the program at the type level. For example, we can use Generalized Algebraic Datatypes (GADTs) [10] to constrain what values can be represented by core data structures. If fewer invalid values can be represented, then the developer can make fewer mistakes. GHC has powerful pattern match redundancy checking for GADTs [6], so the inability to represent invalid values also means that we do not need to handle invalid patterns. This makes the code cleaner and easier to understand. In addition, we can use GHC's type-level strings, lists, and tuples along with the ability to promote data types [14] to express nearly any value at the type level.

We do not aim to prove that *every* aspect about the code is correct. Rather, we use types to verify individual properties. Expressing more properties leads to a higher level of confidence. In some cases, types even guide the programmer to think through edge cases which are not explicitly checked by the compiler (as seen in Section 4.1). Even lightweight applications of dependent types go a long way.

3 Thrift: A Motivating Example

The Thrift Interface Description Language (IDL) [11] is used to define data structures and interfaces for Remote Procedure Calls (RPCs) that can be used across many programming languages. Users can define several types of data structures:

- **Structs** in Thrift are similar to structs in C. A Thrift struct is a data structure with user-defined fields. Each field has an integer identifier which is unique within the struct, a name, and a type.
- **Enums** are simple enumeration types. They can take on one of many values.
- **Typedefs** are named type aliases.

Thrift enables backend services to communicate via strongly-typed APIs. Developers use auto-generated clients in their favorite programming language to query any existing Thrift service. These clients are generated by a Thrift compiler.

```
struct User {
  1: i64 id,
  2: string name,
  3: Pet pet,
}

enum Pet {
  Dog = 0,
  Cat = 1,
}

service MyService {
  User getUser(1: i64 id)
}
```

Figure 1. Example Thrift IDL code

The input to the Thrift compiler is a source file written in the Thrift IDL. Figure 1 shows an example of such source code. In this example, we define an RPC for fetching data about a user given their user ID. The result is a data structure containing three fields including the user's pet which is defined as a Thrift enum. The output of the Thrift compiler is source code in some programming language. The above Thrift code would produce Haskell code that looks roughly like this:

```
data User = User
  { user_id  :: Int
  , user_name :: String
  , user_pet :: Pet
  }

data Pet = Dog | Cat

getUser :: Int → IO User
getUser user_id = ...
```

The Haskell Thrift compiler is used to generate Haskell code from Thrift IDL files. The Thrift IDL files are parsed into an AST, typechecked, and then output as Haskell code. The Thrift compiler internals make extensive use of dependent Haskell in order to guarantee correctness invariants. For example, the typechecked AST must be wellformed. Previous work has been done to implement compilers with encoded invariants in Haskell [5]. However, these compilers were not used in a production setting and therefore they did not demonstrate that these techniques eliminate bugs in practice.

The Sigma codebase relies on millions of lines of code generated by the Thrift compiler in order to fetch additional data needed by abuse detection rules. Bugs in the Thrift compiler could cause the fetched data to take on an incorrect value and therefore cause rules to make incorrect decisions.

Deploying the Haskell Thrift compiler exposed many bugs in the existing C++ implementation which is used to generate Thrift code for languages other than Haskell. These bugs involved accepting ill-typed inputs, infinite looping, mistaken coercions, and ambiguous behavior.

In the following sections we will present a variety of techniques that are deployed in the Haskell Thrift compiler, show the types of bugs that they prevent, and give more examples of how these techniques can be reused for other applications.

4 Constrained Data Structures

Using a combination of GADTs and Promoted Data Kinds [14], we can constrain the values that a data structure can take on. By carefully constructing the data structure upfront, the programmer is forced to think through various edge cases later on.

As part of the compilation process, Thrift code must be typechecked. The Haskell implementation of the Thrift typechecker uses two different Abstract Syntax Tree (AST) representations, an *unresolved* representation for before typechecking and a *resolved* representation for after typechecking.

The implementation centers around the definition of a Thrift type as a GADT which uses the `Status` data kind to distinguish between values that can exist only during certain stages of compilation. A simplified version of this data type can be found in Figure 2. Note that the single quote prefix (eg `'Resolved`) denotes a data constructor that has been promoted to the type level. The data type also has a second type parameter which we will discuss in Section 5.

Base types and collections can be either resolved or unresolved allowing a shared data constructor for both states. Named types must be resolved during typechecking to determine whether they are type aliases, structs, enums, or undefined. The `TNamed` constructor can only be used in an unresolved AST due to its type being `Unresolved`. This ensures that the AST will only contain resolved named types after typechecking.

4.1 Infinitely Recursive Types

In Thrift IDL code, it is possible to create an infinite loop using type aliases:

```
typedef X Y
typedef Y Z
typedef Z X
```

This Thrift program is invalid. The types `X`, `Y`, and `Z` are all defined in terms of each other and therefore none of them can be instantiated as a concrete type. When faced with this input, the C++ Thrift compiler entered into an infinite loop. This happened because the code did not resolve the value of the type alias correctly. It was hard to detect this divergent behavior because Thrift compilation happens as part of a large multi-process build job. Developers were therefore confused as to why the build was hanging. The infinite loop may not even have been in their own Thrift code, but rather in a dependent source file.

```
data Status = Resolved | Unresolved
data Bottom

data Type (u :: Status) (t :: ★) where
  TInt    :: Type u Int
  TBool   :: Type u Bool
  TString :: Type u String
  TList :: Type u t → Type u [t]
  TMap  :: Type u k → Type u v → Type u (Map k v)

  TNamed :: String → Type 'Unresolved Bottom

  TAlias
    :: String
    → Type 'Resolved t
    → Type 'Resolved t
  TStruct
    :: String
    → Schema s
    → Type 'Resolved (StructVal s)
  TEnum
    :: String
    → EnumSchema s
    → Type 'Resolved (EnumVal s)
```

Figure 2. Definition of Thrift Types

Named types can refer to each other, forming a graph structure. The resolution step must be done in topological order with respect to the graph of named types. The C++ implementation did not attempt to resolve type aliases and instead it traversed the pointers indefinitely during code generation.

This is an interesting result because the bug is not explicitly precluded by the resolved named types invariant. Rather, the need for topological sorting was made obvious by the fact that all type aliases must be fully resolved to a concrete type. In a less strongly typed AST, it would be possible for the Thrift compiler to shallowly resolve a type alias to:

```
TAlias "Y" (TNamed "X")
```

This is essentially what the C++ Thrift compiler did. However, that term is not well-typed in Haskell because it mixes resolved (`TAlias`) and unresolved (`TNamed`) data constructors. This inability to mix values made it obvious that types referred to in the type alias must be resolved before the alias itself. In this way, the properties that we encode in the AST force us to think through edge cases even if they are not explicitly checked by GHC.

4.2 Additional Applications: Sync vs Async Rules

The idea of using GADTs and Data Kinds to constrain data type values can be applied to a wide variety of applications outside of compilers. In Sigma, we use this same idea to differentiate between types of rules.

Rules are evaluated in two rounds. First, the *synchronous* rules are run and return a response back to the client. For example, a synchronous rule could return a Tag response which adds additional metadata to a request. More heavyweight classification is run in the *asynchronous* round. These rules can only perform actions after the fact such as logging information.

The key invariant relating to rule types is that synchronous responses can only be returned within synchronous rules. A synchronous response returned by an asynchronous rule would have no effect because the request that it is trying to modify has already finished. Conversely, async responses can be returned anywhere because they make no assumption about the status of the request being processed.

Similar to the Thrift example, rule types are encoded as a data kind and responses are encoded as a GADT. The structure can be seen below where Tag is constrained to be Sync and Log is universally quantified.

```
data RuleType = Sync | Async

data Response (t :: RuleType) where
  Tag :: Response 'Sync
  Log :: Response t
  ...
```

In this scheme, the following rule is not well-typed because it is returns a synchronous response in an asynchronous rule.

```
checkScore :: Double → [Response 'Async]
checkScore score =
  if score > 0.9 then [Tag] else []
```

There was initially no type-level distinction between sync and async responses; Sigma developers needed to rely on a boolean flag in a separate configuration file in order to determine what type of rule they were working with. This became intractable especially in deeply nested helper functions. Adding the type-level distinction caused this invariant to be checked by GHC. This uncovered hundreds of violations in which sync responses were getting silently dropped.

5 Associated Types

We can encode propositions in the type signatures of functions by associating the type parameters of multiple data types.

The GADT representing Thrift types in Figure 2 has a second type parameter of kind ⋆ (the kind of terms). This type parameter does not introduce any invariants on the data structure itself; Thrift types are arbitrarily composable. However, we use the parameter in the Thrift typechecker to guarantee that typechecked constants are wellformed by associating it with the type parameter of a TypedConst.

An UntypedConst is a sum type used in the unresolved Thrift AST to represent the syntax of Thrift constants. A TypedConst is used in the resolved AST and is either an identifier or a Haskell literal of type t.

```
data UntypedConst
  = IntLit Int
  | StrLit String
  | ...

data TypeConst t
  = Identifier String (Type 'Resolved t)
  | Literal t
```

The core typechecking function associates the type parameters of the input Type and the output TypedConst.

```
typecheckConst
  :: Type 'Resolved t
  → UntypedConst
  → Either TypeError (TypedConst t)
```

This guarantees the wellformedness of literals. Consider the following partial implementation of typecheckConst.

```
typecheckConst TInt (IntLit n) = Right $ Literal n
typecheckConst TInt (StrLit s) = Right $ Literal s
```

The first line is correct; we are typechecking a TInt therefore we are allowed to return a Haskell Int. The second line will cause a type error in GHC because returning a String violates the constraints introduced by pattern matching on the TInt data constructor.

While the output must be wellformed, this does not guarantee total correctness. An implementation of this type signature could go wrong in several ways including returning an error on any input or returning 0 for every integer. However, those mistakes are not particularly easy to make as they require the programmer to intentionally insert a new value. In our experience, no bugs of this kind have been introduced. We will see how this mechanism combined with type-level schemas provides strong guarantees in Section 6.

5.1 Additional Applications: Typed Data Fetches

Sigma uses a library called Haxl [9] in order to efficiently handle data fetching by dispatching fetches asynchronously, automatically batching fetches to the same datasource, and caching the results. In order to implement this behavior, data fetches need to be represented as data types rather than simple IO functions. These data types are aggregated into batched IO requests and used as keys in the data cache.

Different data fetches have different return types, therefore each request type is a GADT which associates the request data constructor with its return type. For example:

```
data UserDataRequest a where
  GetName :: Int → UserDataRequest String
  GetPet :: Int → UserDataRequest Pet
```

```
data Schema (s :: [(Symbol, ⋆)]) where
  SNil :: Schema '[]
  SCons
    :: ∀ (name :: Symbol) t s. KnownSymbol name
    ⇒ Type 'Resolved t
    → Schema s
    → Schema ('(name, t) ': s)

data StructVal (s :: [(Symbol, ⋆)] where
  SVNil
    :: Schema '[]
  SVCons
    :: ∀ (name :: Symbol) t s. KnownSymbol name
    ⇒ Type 'Resolved t
    → TypeConst t
    → StructVal s
    → StructVal ('(name, t) ': s)

typecheckStruct
  :: Schema s
  → [(String, UntypedConst)]
  → Either TypeError (StructVal s)
```

Figure 3. Schemas and Values for Structs

The function that performs the fetching, caching, and batching uses the GADT's type parameter to guarantee the output type of the fetch:

```
dataFetch :: DataSource r ⇒ r a → Haxl a
```

DataSource is a type class that defines how to perform the fetch and batch requests and UserDataRequest has an instance of DataSource. The type signature of dataFetch is similar to that of typecheckConst in that both guarantee that any output value is wellformed according to the type of the input. A wrapper function provides a convenient way to fetch data with caching and batching done automatically.

```
getName :: Int → Haxl String
getName userId = dataFetch $ GetName userId
```

6 Type-Level Schemas

The Thrift compiler must check wellformedness for structs and enums using the associated types in the typecheckConst function. However, these types are created by users and therefore there is no standard Haskell type that we can associate with structs or enums. Instead, we use type-level schemas to dynamically build representations of those types.

For structs, the wellformedness property states that all named fields are present and well-typed. The schema type is therefore a type-level list of field names (of kind Symbol) and types (of kind ⋆). The structure of schemas and struct literals is shown in Figure 3. The function for typechecking struct literals associates the type parameter of the Schema with that of the StructVal in order to ensure that the result is wellformed. For example, the schema for the User struct in Figure 1 is as follows:

```
SCons @"id" TInt
 (SCons @"name" TString
  (SCons @"pet" (TEnum ...) SNil)) ::
 Schema
  '[("id", Int), ("name", String), ("pet", Pet)]
```

By construction, any StructVal with the same schema type must contain well formed values for the "id", "name", and "pet" fields. We can now complete the definition of typechecking struct literals:

```
typecheckConst (TStruct _ schema) (MapLit fields) =
  Literal <$> typecheckStruct schema fields
```

Figure 4 gives an outline for how enums are typechecked. Typechecking an enum generates a proof that the enum's name is an element of the desired schema. The proof object is a singleton [3] which essentially expresses the index of the relevant element in a type-level list. This is the opposite of what we do for structs; instead of proving that *all* fields are present, we need to prove that *any* member is present.

```
data EnumSchema (s :: [Symbol]) where
  ESNil :: EnumSchema '[]
  ESCons
     :: ∀ (name :: Symbol) s. KnownSymbol name
    ⇒ Proxy name
    → EnumSchema s
    → EnumSchema (name ': s)

data EnumVal (s :: [Symbol]) =
  ∀ n. EnumVal String (MembershipProof n s)

data MembershipProof x xs where
  PHere :: MembershipProof x (x ': xs)
  PThere
    :: MembershipProof x xs
    → MembershipProof x (y ': xs)

typecheckEnum
  :: EnumSchema s
  → Proxy name
  → Maybe (MembershipProof name s)
typecheckEnum ESNil _ = Nothing
typecheckEnum (ESCons name s) name' =
  case eqT name name' of
    Just Refl → Just PHere
    Nothing → PThere <$> typecheckEnum s name'
```

Figure 4. Enum Schemas and Typechecking

Intuitively, the MembershipProof is a proof by induction that an element x is contained in a list. The base case (PHere) states that if x is the head of the list, then x is contained within the list. We then use an inductive step (PThere) to conclude that if x is contained in some list xs, then x is also contained in the list $y : xs$.

When typechecking enums, we build a MembershipProof by iterating through the schema to find the element that we are looking for. We take in the name of the identifier that we are typechecking as a term-level String. In order to get a type-level representation of the identifier, we use someSymbolVal from the GHC.TypeLits library. This is necessary because Haskell is not truly a dependently typed language and therefore there is a phase separation between types and terms.

The eqT function (from Data.Typeable) checks whether two type-level strings have the same type and Refl is a proof that the types are equal. Pattern matching on the ESCons and Refl constructors allows GHC's constraint solver to resolve s to (name ': s') which satisfies the constraint of the PHere constructor, completing the proof. If the names are not equal (the Nothing case), then we recurse deeper into the schema. Returning a proof verifies that the identifier we are checking must be a valid enum value. We now complete the typechecking function by packing the proof into an EnumVal.

```
typecheckConst (TEnum schema) (Ident symbol) =
  case someSymbolVal symbol of
    SomeSymbol name →
      case typecheckEnum schema name of
        Just pf → Right $ Literal $ EnumVal symbol pf
        Nothing → Left $ TypeError $ ...
```

The act of building the proof object introduces additional runtime and space complexity. We need $O(n)$ space to store the proof whereas an unverified version of the code would not need any additional space. We also traverse the schema linearly rather than using a set to simply check for membership in constant time. We could use a type-level set, but it would add significant complexity in building up a proof object compared to using a simple list. In practice, enums rarely have more than a handful of constructors and performance is not an issue.

6.1 Common Bugs with Thrift Enums

Although enums are a seemingly simple concept, the semantics of Thrift enums give rise to numerous bugs. The C++ Thrift compiler essentially treats enums as integers. This means that there is no checking done to ensure that an integer is a valid value for a given enum.

In Thrift, it is legal to use integer literals for enums. The following example is valid Thrift IDL code and the last line is equivalent to setting value equal to B. However, it would not be legal to set value equal to 3, because the enum X has no member with value 3.

```
enum X {
  A = 0,
  B = 1,
  C = 2,
}

const X value = 1
```

This case was overlooked in the C++ Thrift typechecker whereas the bug was not present in the Haskell implementation because the Haskell implementation needed to find a name associated with the numeric value which is provably part of the schema. While this bug is not desirable, worse yet is the ability to set a constant of one type equal to an enum value of a different type.

```
enum Status {
  Ok = 0,
  Error = 1,
}

enum Result {
  ERROR = 0,
  OK = 1,
}

const Status error_status = ERROR
```

In this example, multiple enums define error states with different capitalization. It is easy for the programer to accidentally use the wrong one in Thrift IDL code. While programmers may think that the value of error_status is an error state, they are unwittingly setting the value to Ok. Although this is a toy example, a real bug of this nature was discovered in production code because the file failed to pass the Haskell Thrift typechecker. This type of silent coercion could be used intentionally, but the instances found in production were bugs.

Further ambiguity arises from the semantics of enum scoping. In Thrift, symbols imported from other modules must be qualified using the module names. Enums on the other hand can optionally be qualified in order to disambiguate values from different enums. This means that a value called X.A could either refer to a value A from a module called X.thrift, or it could be a value A from an enum called X that is defined in the local module. Once again, a bug of this nature was discovered thanks to the Haskell Thrift typechecker.

The fact that the Haskell Thrift typechecker had fewer bugs is no coincidence. Expressive types force the programmer to be more rigorous in the implementation. We *cannot* accept arbitrary integers or identifiers as enum values because we need a valid name with which to construct a MembershipProof. The programmer could get invent a name, but that would entail knowingly inserting an incorrect value. The goal of using dependent types is to guide the programmer towards a correct implementation, not to fully verify the code.

6.2 Additional Applications: Schematized Inputs

The Sigma API uses JSON input maps to pass data to abuse detection rules. The API for accessing inputs inside of rules is:

```
lookupInput :: FromJSON a ⇒ Text → Haxl a
```

Haxl is a monad which is primarily used for asynchronous data fetching [9]. In this case, we are only using it as a reader monad to inspect the input map and to represent exceptions.

The lookupInput function can fail in one of two ways. First, the input key may not be present in the input map. Second, the key may be present, but it may have a different JSON type than the programmer requested. In practice, the first failure type is more prominent.

The obvious solution to this problem would be to switch to using strongly typed inputs. In this scheme, each rule would define a Haskell datatype as its input type. However, this would greatly reduce opportunities for code sharing. If each rule had a separate input type, there would be no way to express that some common helper function requires a subset of keys that are present in many rules (Haskell does not support row polymorphism).

To get the best of both worlds, we use a type-level schema on top of strongly typed inputs in order to enable code sharing. The new input lookup API is:

```
get
  :: ∀ (key :: Symbol) ty input. Has key ty input
  ⇒ ty
```

As opposed to lookupInput, get is pure indicating that it cannot fail (pure exceptions are not used in the Sigma codebase). This is because the Has constraint ensures that the key is available in input with type ty. Note that get takes no term arguments. To specify which key to look up, the programmer uses a visible type application [4].

```
getUserId :: Has "UserId" Id input ⇒ Id
getUserId = get @"UserId"
```

Type application is a foreign concept to most Sigma developers, however it is natural to use because it looks much like what they are used to when they use lookupInput.

```
getUserId' :: Haxl Id
getUserId' = lookupInput "UserId"
```

Another option would have been to use a Proxy, however in practice we found that the type application was easier to understand for Sigma developers who are not familiar with advanced Haskell type system features. Using type-level schemas, we were able to construct a safe API with the same usability and portability as the untyped API.

7 Conclusion

The increasing complexity of modern codebases makes it difficult for engineers to reason about correctness. Using advanced techniques to express invariants in the type system greatly enhances our ability to write correct code. These techniques have traditionally only been explored in academic settings, however there is evidence that using them in the software engineering industry will lead to great improvements in code correctness.

This work shows promising results from using Dependent Haskell. We were able to eliminate errors in diverse real-world software projects including compilers and abuse detection rules. These errors include accepting ill-typed inputs in a typechecker, failing due to missing inputs, and incorrectly coercing values. It therefore makes sense to incorporate dependent types in future Haskell software projects.

Potential downsides of using dependent types include longer compile times due to increased constraint solving and cryptic error messages. In practice, we did not find either of these concerns to be an issue.

The majority of software is not written in Haskell; most software is implemented in imperative languages, many of which are not strongly-typed. However, the software industry is trending towards making code statically typed. This includes initiatives to build typecheckers for PHP [12] and Javascript [1]. Programmers are increasingly forced to fix type errors as they write their code; these errors used to manifest themselves at runtime. This experience report provides evidence that it is worthwhile not only to check types statically, but also to continue evolving the expressivity of type systems for mainstream programming languages. Common Haskell paradigms such as higher-order functions, immutability, and control of side effects have already started to show up in imperative languages. Dependent types could be the next big step.

Acknowledgments

I would like to thank Stephanie Weirich for opening my eyes to Dependent Types and for encouraging me to write about my experience. I am also grateful to my colleagues Andrew Farmer, Simon Marlow, and Josef Svenningsson for giving feedback on a draft of this paper and for reviewing my dependently typed Haskell code as well as the anonymous Haskell Symposium reviewers who gave transformative feedback.

References

[1] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):48:1–48:30, 2017.

[2] David Thrane Christiansen, Iavor S. Diatchki, Robert Dockins, Joe Hendrix, and Tristan Ravitch. Dependently typed haskell in industry (experience report). *Proceedings of the ACM on Programming Languages*, 3(ICFP):100:1–100:16, 2019.

[3] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2012.

[4] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. Visible type application. In *Proceedings of the European Symposium on Programming*, pages 229–254, 2016.

[5] Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. *ACM SIGPLAN Notices*, 43(9):75–86, 2008.

[6] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. GADTs meet their match: Pattern-matching warnings that account for GADTs, guards, and laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pages 424–436, 2015.

[7] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed Haskell programming. In *Proceedings of the ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 81–92, 2013.

[8] Simon Marlow. Fighting spam with Haskell. https://engineering.fb.com/security/fighting-spam-with-haskell/, 2015. Accessed: 2020-07-07.

[9] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. *ACM SIGPLAN Notices*, 49(9):325–337, 2014.

[10] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, 2006.

[11] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[12] Julien Verlaguet and Alok Menghrajani. Hack: A new programming language for HHVM. https://engineering.fb.com/developer-tools/hack-a-new-programming-language-for-hhvm/, 2014. Accessed: 2020-07-07.

[13] Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *Proceedings of the ACM on Programing Languages*, 1(ICFP):31:1–31:29, 2017.

[14] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, 2012.