Describing Microservices using Modern Haskell (Experience Report)

Alejandro Serrano
alejandro.serrano@47deg.com
47 Degrees
San Fernando, Cádiz, Spain

Flavio Corpa
flavio.corpa@47deg.com
47 Degrees
San Fernando, Cádiz, Spain

Abstract

We present Mu, a domain specific language to describe and develop microservices in Haskell. At its core, Mu provides a type level representation of schemas, which we leverage in various ways. These schemas can be automatically imported from industry-standard interface definition languages.

Mu uses many of the type level extensions to GHC, and techniques such as (data type) generic programming and attribute grammars. Apart from the description of the library, we discuss a series of shortcomings in current GHC/Haskell, mostly related to the friendliness of the exposed library interface once complex types enter the scene.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Domain specific languages**; • **Theory of computation** → *Type structures.*

Keywords: Haskell, type classes, microservices, type level programming

ACM Reference Format:
Alejandro Serrano and Flavio Corpa. 2020. Describing Microservices using Modern 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/10.1145/3406088.3409018

1 Introduction

In the last decades Haskell as offered by GHC – the de facto standard compiler for Haskell – has outgrown from a simple "Hindley-Milner plus type classes" to give much more expressive power, especially for type level and generic programming. Functional dependencies [29], type families [3, 9], or data type promotion [32], make the world of types at least as interesting as that of terms. More powerful views on data types [15, 16, 18] usually build on those new mechanisms.

Embedded (or internal) domain specific languages (EDSLs) have greatly benefited from these features. Most EDSLs come with rules and invariants that can be enforced by the compiler, if one is able to express them at the type level. Examples abound: information security [22], hardware specifications [14], structure-aware diffing [19], or REST-based services [17].

In this paper we present Mu,[1] a EDSL to develop microservices. The primary goal of Mu is to provide a unified interface to a range of protocols and serialization formats, including gRPC[2] and GraphQL,[3] a goal shared between the different incarnations of the Mu library in different languages. A second goal of the Haskell version is to introduce type safety based on the different "service or schema definition languages" provided by those protocols. You can even take a definition of a gRPC service and then serve it using GraphQL.

To implement Mu we have used several techniques stemming from academia, including the aforementioned generic and type level programming to describe schemas, but also attribute grammars to define what a "server" for such a schema is (Section 2). An important observation is that we can re-use most of the typing information available in a system if we provide the ability to import such information. This is not the only lesson we have learned in our journey, Sections 3 to 6 provide a view on the current status and weaknesses of modern Haskell in a set of areas.

2 Type Level Schemas in Mu

Mu is not a single package, but a constellation of them. In fact, Mu offers libraries for different functional programming languages, including Scala and Haskell. In this paper, whenever we refer to "Mu", we refer to the Haskell version.

The base packages provide a unified language for describing schemas of data – package `mu-schema` – and service definition along with server implementations – package `mu-rpc`. Most of the protocols we support include this separation between data and behavior, Figure 1 shows an example written in Protocol Buffers. The rest of the packages provide the support for different protocols: gRPC with Avro or Protocol

[1] https://higherkindness.io/mu-haskell/
[2] https://grpc.io/
[3] https://graphql.github.io/

Haskell '20, August 27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8050-8/20/08…$15.00
https://doi.org/10.1145/3406088.3409018

```
message PersonRequest { int64 id = 1; }
message Person { string name = 1; int32 age = 2; }
service PersonService {
  rpc getPerson (PersonRequest) returns (Person);
  rpc newPerson (Person) returns (PersonRequest);
}
```

Figure 1. Description of service using Protocol Buffers

```
data TypeDef typeName fieldName
  = DRecord typeName [FieldDef typeName fieldName]
  | DEnum   typeName [ChoiceDef fieldName]
  | DSimple (FieldType typeName)
newtype ChoiceDef fieldName
  = ChoiceDef fieldName
data FieldDef typeName fieldName
  = FieldDef fieldName (FieldType typeName)
data FieldType typeName
  = TNull
  | TPrimitive Type
  | TSchematic typeName
  | TOption (FieldType typeName)
  | TList   (FieldType typeName)
  | TUnion  [FieldType typeName]
```

Figure 2. Schema definition language

Buffers serialization, and GraphQL; in most cases they "just" consume the information from the core packages.

2.1 Describing Terms

Schemas of data serialized in the wire – like *PersonRequest* and *Person* in the example heading this section – are described using the language in Figure 2. Note that although we define types and constructors in the term level, these are intended to be used only *promoted* [32] to the type level. Figure 5a shows how these two types defined in Figure 1 are translated into this language.

As a design choice we leave the types representing names open. Users of the library can decide between using a more stringy-typed approach – using *Symbol*, the promoted version of *String* – or a more strongly-typed approach – by using an enumeration for that specific set of field or type names. Note that even with the former approach type names are checked for existence at compile time.

The data itself is represented (or interpreted) as a *Term* from Figure 3. For example, a record is interpreted into a

```
type family (sch :: Schema t f) :/: (name :: t)
            :: TypeDef t f where
  `[] :/: name = TypeError ...
  (`DRecord name fields `: rest) :/: name
    = `DRecord name fields
  (`DEnum name choices `: rest) :/: name
    = `DEnum name choices
  (other `: rest) :/: name = rest :/: name
data Term (sch :: Schema t f) (t :: TypeDef t f) where
  TRecord :: NP (Field sch) args
          -> Term sch (`DRecord name args)
  TEnum   :: NS Proxy choices
          -> Term sch (`DEnum name choices)
  TSimple :: FieldValue sch t
          -> Term sch (`DSimple t)
data Field (sch :: Schema t f) (f :: FieldDef t f) where
  Field :: FieldValue sch t
        -> Field sch (`FieldDef name t)
data FieldValue (sch :: Schema t f) (t :: FieldType t) where
  FNull      :: FieldValue sch `TNull
  FPrimitive :: t
             -> FieldValue sch (`TPrimitive t)
  FSchematic :: Term sch (sch :/: t)
             -> FieldValue sch (`TSchematic t)
  FOption    :: Maybe (FieldValue sch t)
             -> FieldValue sch (`TOption t)
  FList      :: [FieldValue sch t]
             -> FieldValue sch (`TList t)
  FUnion     :: NS (FieldValue sch) choices
             -> FieldValue sch (`TUnion choices)
```

Figure 3. *Term*s, interpretations of schemas

```
class ToSchema (sch :: Schema typeName fieldName)
               (sty :: typeName) (t :: Type)
               | sch t -> sty where
  toSchema :: t -> Term sch (sch :/: sty)
```

Figure 4. Conversion from Haskell type to *Term*

product[4] of the results of interpreting its fields; the interpretation of a field with type `` `TList t `` is a list of values of the interpretation of *t*; and so on. Figure 5b shows a specific value for *Person* represented as a *Term*. One important case is the reference from one type to another in the same schema, given by the *FSchematic* constructor. In that case we need to find the definition of such type in the whole schema, this is the duty of the (:/:) type family.

[4]The types *NP* and *NS* come from the generics-sop library [6].

```
type Schema
  = `[`DRecord "PersonRequest"
        `[`FieldDef "id"    ('TPrimitive Int64)]
    , `DRecord "Person"`
        `[`FieldDef "name" ('TPrimitive String)
        , `FieldDef "age"  ('TPrimitive Int32)]]
```

(a) Type level schema description

```
authors :: [Term Schema (Schema :/: "Person")]
authors
  = [TRecord  (  Field (FPrimitive "Alejandro")
              :* Field (FPrimitive 32) :* Nil)
    , TRecord  (  Field (FPrimitive "Flavio")
              :* Field (FPrimitive 29) :* Nil)]
```

(b) Example of *Person* as a term

```
data Person = Person {name :: String, age :: Int32}
instance ToSchema Schema "Person" Person where
  toSchema Person {name, age}
    = TRecord  (  Field (FPrimitive name)
               :* Field (FPrimitive age) :* Nil)
```

(c) Conversion to a regular data type

Figure 5. Examples using *PersonService*'s schema

We do not expect programmers to manipulate those *Term*s directly (although using optics and overloaded labels this is feasible, see Section 3.1); instead we provide a pair of type classes to define conversion to usual Haskell types. In Figure 4 we show the definition of the class conversion to *Term*, the one in the other direction is completely symmetric. Notice the use of (:/:) to look up the information associated to the given type name *sty* in the whole schema *sch*. Figure 5c shows the instance corresponding to our *Person*.

What we have here is nothing else than *(data type) generic programming*: we define the universe of our types (Figure 2), their interpretation (Figure 3), and the conversion to the generic representation (Figure 4). The style we follow here is closer to the SOP representation [6], extended for mutual recursion [18] in the schema. By using "generic generic programming" [16] we can then turn the representation in terms of the built-in *Generics* module in GHC [15] into our own.

Importing schemas. At the beginning of this section we have shown the definition of a service, including the schema of the data, in the Protocol Buffers language. This kind of descriptions is already available for many services, but it is quite tiresome to translate them to Mu's type level language. For that reason, we have implemented several *importers* which read a file in one of the supported formats, and translates it to types, using Template Haskell [27].

```
type PersonService
  = `[  `Method "getPerson"
          `[`ArgSingle `Nothing "PersonRequest"]
          ('RetSingle "Person")
      , `Method "newPerson"
          `[`ArgSingle `Nothing "Person"]
          ('RetSingle "PersonRequest")]
```

(a) Type level service description

```
personServer
  = getPerson :<|>: newPerson :<|>: H0
  where
    getPerson (PersonRequest ident) = do
      p ← execute "SELECT * FROM Person ..."
                  [":id" := ident]
      pure p
    newPerson (Person name age) = do
      ident ← execute "INSERT INTO Person ..."
      pure (PersonRequest ident)
```

(b) Handlers for *PersonService*

Figure 6. Examples using *PersonService*

This gives us one additional freedom: since we know almost no user of the library specifies the types in Figure 2 directly, we can change them without a lot of fuss. In fact, we have done it on each new major release of the library.

Automatically importing from a schema language helps us to keep in sync with the contract agreed on a system, in which several parts need to communicate. This is a well-known pain-point in industry, which has prompted the development of solutions such as Confluent's Schema Registry for Apache Kafka.[5]

2.2 Describing Services and Servers

The description of services, that is, the set of functionality that a client may request from a server, is encoded in the type level, using the same techniques as the schemas. Figure 6a shows how to encode *PersonService* from Figure 1 – notice the similarities with Figure 5a – we have a list of methods, each with their corresponding list of arguments and return type. The additional *ArgSingle* and *RetSingle* are required to account for the possibility in most RPC frameworks to return not just one, but a stream of values as result of a method.

The interesting bit in this case is what an *interpretation* of that definition means: instead of data, what we need is the *behavior* associated to the service. In other words, an implementation. Figure 7 shows an extract of the definition of *handlers* for a service made up from a set of *methods*. *HandlersT* follows the familiar shape of a heterogeneous list,

[5] https://docs.confluent.io/current/schema-registry

```
data HandlersT inh methods m hs where
  H0     :: HandlersT inh `[] m `[]
  (:<|>:) :: Handles as r m h
          => (inh -> h) -> HandlersT inh ms m hs
          -> HandlersT inh (Method n as r `: ms) m (h `: hs)
instance (FromRef ref t, Handles args ret m h,
          handler ~ (t -> h))
  => Handles (ArgSingle aname ref `: args) ret m handler
instance (FromRef ref t, Handles args ret m h,
          handler ~ (ConduitT () t m () -> h))
  => Handles (ArgStream aname ref `: args) ret m handler
instance (handler ~ m ())
  => Handles `[] `RetNothing m handler
instance (ToRef ref v, handler ~ m v)
  => Handles `[] (RetSingle ref) m handler
instance (ToRef ref v,
          handler ~ (ConduitT v Void m () -> m ()))
  => Handles `[] (RetStream ref) m handler
```

Figure 7. Handlers for services

each element being one such handler. The type required to implement a method changes depending on the number of arguments, the way they are received – as a single value or as a stream of values –, and the information they must return – now to a single or a stream of values we add the possibility of returning nothing. The *FromRef* and *ToRef* type classes allow users of the library to choose between consuming and providing *Term*s, or use a regular Haskell type tied to the schema using *FromSchema* and *ToSchema* as discussed in Section 2.

In the case of *PersonService*, the implementation[6] given in Figure 6b only receives or returns single results. In addition, we make use of the regular Haskell types introduced in Figure 5c.

In order to support GraphQL, the execution of a service – called a *resolver* in that case – may continue with another service, if we need to request further information. We have modeled this fact after *attribute grammars* [5, 30], where each service receives the result of its "parent" as an inherited attribute (hence the *inh* parameter in *HandlersT*) and synthesizes the result.

2.3 Conclusion

The main lesson we have learned here is that by rephrasing some well-known ideas in academia in terms of well-known technologies in industry we can bridge the gap between these two worlds. In this case, by using type level and generic programming to describe data schemas.

[6]The code in the implementation does not work, it is just pseudocode whose database access is inspired by the sqlite-simple package.

There is also lots of information about the contract of an interface laying around: database schemas, OpenAPI service definitions, Kafka Registry schemas, among others. As proponents of a strongly-typed discipline, we should take advantage of that information and try to integrate as much of it as possible in our code.

3 Lesson #1: Type Applications vs. Labels

Schemas describe in depth both the shape of the data exchanged by the services, and the arguments and result types of the different methods, as we have discussed in Section 2. In many cases we can leverage this information to directly execute or compute. The main two examples in Mu are calling a method – that is, working as a *client* instead of as a server –, and inspecting or modifying a field.

Our first approach was to use *visible type application* [10] to specify the method or field in question. This usage is not novel, it had been reported as a way to generically derive traversals [13]. As an example, Mu provides a *gRpcCall* which given the service description, the method name, and any required arguments, issues a call to the corresponding gRPC server. No further information is required, since serialization and communication code can be derived from that piece of type level information.

```
response :: GRpcReply Person
  <- gRpcCall @PersonService @"getPerson" client req
```

Having both *PersonService* and `"getPerson"` at hand, the library knows in which possible forms the request parameters may be provided. Note that we still need to annotate the return type, an issue we shall return to in Section 6. The additional *client* argument represents the connection to the server, including the server and port to connect to and the route to the endpoint, among other data.

Although quite direct, we found a number of problems with exposing the client API in this form:

1. We are forced to provide arguments in a specific order – first type arguments, then value arguments – unless we require the use of the *RankNTypes* extension. In the example above, it is unsatisfactory to provide the *client* parameter after the name of the method to be called; one would expect going from general to particular – connection, service, method, parameters.
2. The syntax feels alien to many Haskellers. For example, the name of the method appears after the @ symbol and wrapped in quotes.

The usual workaround – requiring *Proxy* values instead of using type application – does not really solve the problem. The programmer has to write more boilerplate, and still has to mention the types:

```
response :: GRpcReply Person
  <- gRpcCall client (Proxy :: Proxy Service)
                     (Proxy :: Proxy "getPerson") req
```

The first actual solution came in the form of records which were "filled" using data type generic programming. The programmer declares a data type with a single constructor, and fields for every possible method in the service:

```
data Call = Call
  { getPerson :: PersonRequest → IO (GRpcReply Person)
  } deriving Generic
```

The *buildService* function generates one value of that type given the type level information and the connection to the server. The programmer can then access each of the methods using regular record syntax.

```
do svc ← buildService@PersonService client
   response ← getPerson svc req
```

This solution is at least easier to automate; with some Template Haskell the shape of the record can be derived from the service definition. Alas, we end up having to replicate the names and expected types of every method in the service.

3.1 Optics and Overloaded Labels

The optics package[7] provides lenses, prisms, traversals, and other kinds of composable data accessors following an *opaque* design. That means that, in contrast to exposing the underlying (higher-rank) types like the lens package does, optics are wrapped into **newtype**s. The main advantage is improved error reporting.

In our case, we took advantage of a small module integrating optics with the *OverloadedLabels* extension [11]. This extension builds on the *IsLabel* type class, defined as:

```
class IsLabel (x :: Symbol) a where
  fromLabel :: a
```

GHC then replaces any usage of *#thing*[8] with a call of the form *fromLabel* @"thing" in a similar vein to how numeric literals are transformed into calls to *fromInteger*.

To solve the problem of gRPC clients, we use this mechanism to provide a set of *getters* – lenses with no setter – from *GRpcClient* to the various methods in the corresponding service. Our previous call now reads a bit simpler, with the client in first position, then the name of the method, and finally the request parameters.

```
client ^. #getPerson $ req
```

Once you get this hammer, everything looks like a label. Another place in which we have introduced overloaded labels is the access to fields of a *Term* without the need of creating an intermediate Haskell type. The variety of optics helps us here: for record fields we provide lenses, but for enumerations and unions we provide prisms instead.

[7] https://github.com/well-typed/optics

[8] At the moment of writing, those labels cannot start with a capital letter, but there's an on-going GHC proposal to lift that restriction.

The brighter future. Using overloaded labels still introduces some alien syntax, namely the combination ^.# before the name of the method. The *RecordDotSyntax* proposal [20] introduces new syntax for getters and setters, and does so in an extensible way, as *OverloadedLabels*. As a result, in a near future the code above could become:

```
client.getPerson req
```

Such an interface is quite intuitive for most programmers.

At this point we want to stress one important design decision of both *OverloadedLabels* and *RecordDotSyntax*, which is sometimes absent from the on-going debate around these features. The use of a type class provides a path to define "virtual" fields or labels, not backed up by or representing any actual data, as we do with gRPC methods.

4 Lesson #2: GADTs Provide a Hard API

We make heavy use of GADTs in the definition of terms following a schema and servers implementing a given service, as we have discussed in Section 2.1. As in the case of Servant [17], the core of the server API consists of a way to join the different handlers. In the case of Mu, the order in which those handlers should appear must coincide with the order in which they appear in the service definition; other libraries normalize the descriptions beforehand to order the fields by name [1].

$$handler_1 :\langle|\rangle: handler_2 :\langle|\rangle: \ldots :\langle|\rangle: H_0$$

Heterogeneous lists may pose no challenge for a experienced type level Haskell programmer; but they do pose a challenge to less experienced users of our library. A less experienced user is likely to make some small mistakes. She may forget to add the final $:\langle|\rangle: H_0$ in the chain, or add a new method in the wrong place, or swap the order of two handlers. Such tiny mistakes can cause a cascade of quite frightening and uninformative error messages, which overwhelm her and create a stumbling block.

Workaround using tuples. As powerful as they are, heterogeneous lists pose a challenge to less experienced users. In the Mu library we have turned into using the main heterogeneous data structure every beginner is aware of: tuples. Converting between them is straightforward.

```
class ToHList p l | p → l where
  toHList :: p → HList l
instance ToHList (a, b)    `[a, b]    where
  toHList (x, y)    = x :⟨|⟩: y :⟨|⟩: H0
instance ToHList (a, b, c) `[a, b, c] where
  toHList (x, y, z) = x :⟨|⟩: y :⟨|⟩: z :⟨|⟩: H0
  -- and as many as you wish
```

As a result, the handlers for *PersonService* can be rewritten as follows:

```
personServer
  = toHList (getPerson, newPerson)
  where ...    -- as before
```

One disadvantage of this approach is that we are limited by the instances of *ToHList*, and ultimately by the length of the tuples provided by the implementation.

Associating names with handlers. Tuples solve the problem of manually writing (:⟨|⟩:) and H_0, but the API still requires matching the order of the handlers with the declaration of methods. The second part of the solution: provide a name-based interface, in which the programmer explicitly matches handlers with the method they implement.

```
server (method @"newPerson" newPerson
       , method @"getPerson" getPerson)
```

This adds some repetition to the code. However, we found this repetition to be quite benefitial: the compiler can still check that all methods have been implemented, and the programmers gets the ability to order methods as they wish, instead of being forced by the type level information.

5 Lesson #3: Type Level is Still Lacking

Mu uses type level schemas and service definitions extensively (Section 2). In order to put them in use, lots of computation need to happen at compile time – in the type level: can this Haskell type be turned into that schema type? Are handlers for all the methods in the service readily available? In this section we look at some of the lessons we have learned while implementing those features.

5.1 Type Classes Live at the Top Level

Multi-parameter type classes [29] and type families [3] are the two GHC features which provide *type level computation*. The former is usually described as having more "relational" style, whereas the latter feels more "functional".

If one inspects the source code of Mu, though, it will be hard to find a declaration of a type family. The main reason is that type classes provide elaboration – that is, term level code can be generated as produce of class resolution – whereas type families do not [26]. We make heavy use of that mechanism: for example, finding the handler for a method in a server definition produces the code which runs that same handler whenever a request is made for it.

Type classes that inspect a complex piece of data – like our service definitions – end up "calling" other type classes which take care of the sub-pieces. This is not a problem on itself, after all chunking functionality in all sub-functions is a good engineering practice. The problem is that type classes are always defined at the top level – there is no notion of *local type class*[9] – so every parameter in the computation has to be carried over through all those classes. That is, whereas at the term level local bindings can use the surrounding environment,

```
f x y = case x of Blah → g [ ]
                  Bluh → ...
  where g z = use x
```

the parameters have to be threaded when using type classes,

```
class F x y
instance G x `[ ] ⇒ F Blah y
instance ...      ⇒ F Bluh y

class G x z
```

Note also that we are required to introduce a new type class G in the top level environment. That pollutes such environment, whereas in the term level version the name g is simply used locally. Furthermore, the definition of that type class include much more boilerplate that the term counterpart.

The more information one needs to consult, the more the problem is exacerbated. As an extreme example, the GraphQL layer for Mu defines the following type class,

```
class RunMethod m p whole chn sname ms hs
```

where every parameter except the last two are just "environment".

Backpack. Much of the type level information threaded through these type classes stays constant within the scope of an application. For example, most people would only write a server for one single service definition per application, not several of them. Instead of parametrizing each single class, we could parametrize the entire *module* using a Backpack mix-in [31]. Alas, Backpack has not been widely adopted. For that reason we have not pursued that path further, since that would introduce an entry barrier for potential users.

5.2 Lack of Abstraction Mechanisms

The language for type level programming in GHC is essentially first order. In other words, no *map* over type level lists, no generic traversals over other promoted data types. This problem has already been reported for type families [12], alongside with a solution: *matchable arrows*. In theory, type classes should require a simpler approach, since they do not suffer from the problem of appearing saturated, as type families do. In addition, *quantified class constraints* [2] extend the power of type class resolution.

Unfortunately, in practice one stumbles upon several rocks: (1) inference works badly, so in many cases we need to use manual type application; and (2) type classes may appear unsaturated, but there are no combinators to reorder the parameters, like we can do in the term level:

```
flip :: (a → b → c) → b → a → c
flip f y x = f x y
```

[9] Local instances have been proposed [7] but have never made into GHC.

The order in which one defines the type class becomes thus extremely important. If one needs to partially apply that class in two different ways, the bets are off. In the specific case of Mu, we end up duplicating the code which traverses schemas or service definitions.

6 Lesson #4: Ambiguous Types

Type error messages are a salient part of a compiler's user experience. There is extensive literature about discussing this aspect of GHC [23, 24, 33] or other functional languages in general [4, 21, 34]. Almost all of this research is focused on the kind of errors stemming from conflicting information, like passing *True* to a function that expects an *Integer*. Little attention has been paid to *ambiguity*.

As a particular example, suppose we take a server definition where some of the implementations are given by $\bot$, as is often done to mark parts of the program yet to be written.

$$\begin{aligned} &\mathit{server}\ (\mathit{method}\ @\texttt{"method1"}\ \mathit{handler}_1 \\ &\quad , \mathit{method}\ @\texttt{"method2"}\ \bot) \end{aligned}$$

We are greeted with a set of error messages similar to:[10]

```
Ambiguous type variable 't1'
prevents the constraint
'(FromSchema <bigtype> "type" t1)'
from being solved.
Probable fix: use a type annotation
to specify what 't1' should be.
These potential instances exist:
instance forall typeName fieldName t ...
-- Defined in 'Mu.Schema.Class'
... plus three instances involving
out-of-scope types
```

The problem here is that by not fixing the type of the handler, the compiler is not able to resolve the serialization from and to Mu's *Term*s to Haskell types.

The compiler is correct in rejecting this program: the variable is in fact ambiguous and the code should not be accepted until such ambiguity is resolved. On the other hand, the error messages are quite daunting, and there is no way to customize them, in contrast to *TypeError* constraints available for conflicts. It may be that, when we think about this problem, we think of "clearly wrong" programs such as:

$$\begin{aligned} &f :: (\mathit{Read}\ a, \mathit{Show}\ a) \Rightarrow \mathit{String} \to \mathit{String} \\ &f = \mathit{show} \circ \mathit{read} \end{aligned}$$

But once complex type level computation enters the game, it becomes easier to write terms which do not fix their types (or kinds) completely. This in turn stops class constraints from resolving and type families from reducing, in a sort of domino effect. Unfortunately, although some research has been done on this problem [25, 28], none of it has reached the GHC compiler (yet).

[10] Slightly edited for readability.

6.1 Implicit Kind Quantification

Kind polymorphism[11] brings parametric polymorphism to type level programming. However, that kind polymorphism may not be directly visible in the code; GHC may introduce inferred type variables [10] while inferring the most general kind for a signature or type class.

Unfortunately, such general inference is a double-edged sword. An instance may not be matched due to some unknown kinds, which are not even visible in type errors. Inference may lead to a "too general" type when we intended the kinds of two of the arguments to coincide. The following code, extracted from Mu's gRPC client support, shows that sometimes we need to *monomorphize* the kinds to make the compiler accept our code:

$$\begin{aligned} &\textbf{instance forall}\ (\mathit{sch} :: \mathit{Schema}\ \mathit{Symbol}\ \mathit{Symbol}) \\ &\qquad\qquad (\mathit{sty} :: \mathit{Symbol})\ (r :: \mathit{Type}) . \\ &\quad \cdots \Rightarrow \mathit{GRpcInputWrapper}\ \texttt{`}\mathit{MsgAvro}\ (\texttt{'}\mathit{SchemaRef}\ \mathit{sch}\ \mathit{sty})\ r \end{aligned}$$

Had we left out the explicit quantification, it would have been inferred to be:

$$\textbf{instance forall}\ (\mathit{sch} :: \mathit{Schema}\ t\ f)\ (\mathit{sty} :: s)\ (r :: \mathit{Type})\ \ldots$$

Note how the kinds of *sch* and *sty* mention *distinct* kind variables, so one more kind variable needs to be resolved in order to know whether this instance matches. This might lead to more ambiguous kinds, as discussed above.

This behavior seems to have surprised more than us. Starting with version 8.10, an accepted GHC proposal [8] introduces changes to kind quantification, making it a bit "less implicit".

7 Conclusion

We have presented Mu, a library for developing microservices in Haskell. Its design has been influenced by several ideas stemming from academia: data schemas are represented at the type level using generic programming techniques, complex servers are defined as attribute grammars. We have also identified some weaknesses of the current programming experience in GHC/Haskell when making use of those language extensions.

Acknowledgments

We would like to thank 47 Degrees for sponsoring the development of the library, and providing interesting use cases to test our designs, and to the Haskell Symposium reviewers for their suggestions.

[11] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#kind-polymorphism

References

[1] Lennart Augustsson and Mårten Ågren. 2016. Experience Report: Types for a Relational Algebra Library. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. ACM, New York, NY, USA, 127–132.

[2] Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. ACM, New York, NY, USA, 148–161.

[3] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. ACM, New York, NY, USA, 241–253.

[4] Sheng Chen and Martin Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 583–594.

[5] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. 2000. First-class Attribute Grammars. *Informatica (Slovenia)* 24, 3 (2000).

[6] Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (Gothenburg, Sweden) *(WGP '14)*. ACM, New York, NY, USA, 83–94.

[7] Atze Dijkstra and S. Doaitse Swierstra. 2005. *Making Implicit Parameters Explicit.* Technical Report UU-CS-2005-032. Department of Information and Computing Sciences, Utrecht University.

[8] Richard A. Eisenberg. 2018. GHC Proposal: Treat kind variables and type variables identically in `forall`. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0103-no-kind-vars.rst

[9] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 671–683.

[10] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 229–254.

[11] GHC Development Team. 2019. `OverloadedLabels`. https://gitlab.haskell.org/ghc/ghc/-/wikis/records/overloaded-record-fields/overloaded-labels

[12] Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. 2019. Higher-Order Type-Level Programming in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 102 (July 2019), 26 pages.

[13] Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic Deriving of Generic Traversals. *Proc. ACM Program. Lang.* 2, ICFP, Article 85 (July 2018), 30 pages.

[14] Jan Kuper, Christiaan Baaij, and Matthijs Kooijman. 2010. Exercises in Architecture Specification Using CLaSH. In *Proceedings of the 2010 Forum on specification & Design Languages, FDL 2010, September 14-16, 2010, Southampton, UK*, Adam Morawiec and Jinnie Hinderscheit (Eds.). ECSI, Electronic Chips & Systems design Initiative, 178–183.

[15] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (Baltimore, Maryland, USA) *(Haskell '10)*. ACM, New York, NY, USA, 37–48.

[16] José Pedro Magalhães and Andres Löh. 2014. Generic Generic Programming. In *Practical Aspects of Decl. Lang.*, Matthew Flatt and Hai-Feng Guo (Eds.). Springer International Publishing, Cham, 216–231.

[17] Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. 2015. Type-Level Web APIs with Servant: An Exercise in Domain-Specific Generic Programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. ACM, New York, NY, USA, 1–12.

[18] Victor Cacciari Miraldo and Alejandro Serrano. 2018. Sums of Products for Mutually Recursive Datatypes: The Appropriationist's View on Generic Programming. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development* (St. Louis, MO, USA) *(TyDe 2018)*. ACM, New York, NY, USA, 65–77.

[19] Victor Cacciari Miraldo and Wouter Swierstra. 2019. An Efficient Algorithm for Type-Safe Structural Diffing. *Proc. ACM Program. Lang.* 3, ICFP, Article 113 (July 2019), 29 pages.

[20] Neil Mitchell and Shayne Fletcher. 2020. GHC Proposal: `RecordDotSyntax`. https://github.com/ghc-proposals/ghc-proposals/pull/282

[21] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 525–542.

[22] Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 280–288.

[23] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 672–698.

[24] Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling Expression-Level Type Errors by Type-Level Programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages* (Bristol, United Kingdom) *(IFL 2017)*. ACM, New York, NY, USA, Article 2, 15 pages.

[25] Alejandro Serrano and Jurriaan Hage. 2019. A compiler architecture for domain-specific type error diagnosis. *Open Computer Science* 9, 1 (2019), 33 – 51.

[26] Alejandro Serrano, Jurriaan Hage, and Patrick Bahr. 2015. Type Families with Class, Type Classes with Family. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. ACM, New York, NY, USA, 129–140.

[27] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. ACM, New York, NY, USA, 1–16.

[28] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2006. Type Processing by Constraint Reasoning. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.

[29] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. 2007. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.* 17, 1 (Jan. 2007), 83–129.

[30] Wouter Swierstra. 2005. Why Attribute Grammars Matter. *The Monad Reader* 4 (2005).

[31] Edward Z. Yang. 2017. *Backpack: Towards Practical Mix-in Linking in Haskell.* Ph.D. Dissertation.

[32] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(TLDI '12)*. ACM, New York, NY, USA, 53–66.

[33] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 12–21.

[34] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 18 (Aug. 2017), 47 pages.