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.

CSC 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:

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, 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 and GraphQL, 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/
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
  | 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) :: (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. Terms, 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

product⁴ 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.

⁴The types NP and NS come from the generics-sop library [6].
We do not expect programmers to manipulate those Terms 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].

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,

\[\text{type} \text{PersonService} = [\text{'Method} \text{"getPerson"} \to \text{[ArgSingle}'\text{Nothing} \text{"PersonRequest"}]\to \text{(RetSingle}\text{"Person"})\to \text{'Method}\text{"newPerson"} \to \text{[ArgSingle}'\text{Nothing} \text{"PersonRequest"}]\to \text{(RetSingle}\text{"PersonRequest"})\]

(a) Type level service description

\[\begin{align*}
\text{personServer} & = \text{getPerson} : (\langle\rangle) ; \text{newPerson} : (\langle\rangle) : H_0 \\
\text{where} \\
\text{getPerson} & (\text{PersonRequest} \text{ident}) = \text{do} \\
& p \leftarrow \text{execute} \text{"SELECT * FROM Person ..."} \to \text{["\text{:id}" := \text{ident}]}
\end{align*}\]

\[\begin{align*}
\text{newPerson} & (\text{Person name age}) = \text{do} \\
& \text{ident} \leftarrow \text{execute} \text{"INSERT INTO Person ..."} \to \text{pure} \text{(PersonRequest ident)}
\end{align*}\]

(b) Handlers for PersonService

Figure 6. Examples using PersonService

5https://docs.confluent.io/current/schema-registry
The main lesson we have learned here is that by rephrasing database access is inspired by the sqlite-simple package.

these two worlds. In this case, by using type level and generic technologies in industry we can bridge the gap between some well-known ideas in academia in terms of well-known 2.3 Conclusion

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 Terms, 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 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.

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:

```haskell
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.

```haskell
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 newtypes. 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:

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

GHC then replaces any usage of `#thing`\(^6\) 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.

```haskell
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>
\[^6\]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:

```haskell
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\[^1\].

```haskell
handler1 :⟨⟩; handler2 :⟨⟩; ... :⟨⟩; 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 `⟨⟩`; `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.

```haskell
class ToHList p l | p → l where
toHList :: p → HList l
instance ToHList (a, b) → [a, b] where
toHList (x, y) = x :⟨⟩; y :⟨⟩; H_0
instance ToHList (a, b, c) → [a, b, c] where
toHList (x, y, z) = x :⟨⟩; y :⟨⟩; z :⟨⟩; H_0 -- and as many as you wish
```

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

\[
\text{flip} :: (a \to b \to c) \to b \to a \to c
\]

\[
\text{flip } f \ y \ x = f \ x \ y
\]
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 ⊥, as is often done to mark parts of the program yet to be written.

\[
\text{server (method @"method1" handler, \ method @"method2" \⊥)}
\]

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:

\[
\text{instance forall typeName fieldName t ... -- Defined in 'Mu.Schema.Class'}
\]

... 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 Terms 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:

\[
f :: (\text{Read a, Show a}) \Rightarrow \text{String} \rightarrow \text{String}
\]

\[
f = \text{show} \circ \text{read}
\]

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).

6.1 Implicit Kind Quantification

Kind polymorphism¹¹ 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:

\[
\text{instance forall (sch :: Schema Symbol Symbol) (sty :: Symbol) (r :: Type).}
\]

\[
\cdots \Rightarrow \text{GRpcInputWrapper MsgAvro (SchemaRef sch sty) r}
\]

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

\[
\text{instance forall (sch :: Schema t f) (sty :: s) (r :: Type) ...}
\]

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.


¹⁰Slightly edited for readability.
References


