Experience Report: Statically-Typed Server APIs

Eric Mertens Iavor S. Diatchki
Galois Inc.
{emertens,diatchki}@galois.com

Abstract

In this paper, we present a technique for implementing statically typed application APIs by making an extensive use of Haskell's type system. We developed the technique as a part of a larger project that aims to produce tools for high-assurance web collaboration.

1. Introduction

In many web applications, the generation and interpretation of URLs is related only by convention. It is the programmer's responsibility to ensure that an application is always generating requests on behalf of a user that it will eventually be able to handle. This can be a problem particularly while the functionality of an application is still actively being developed.

In this paper, we describe how we used Haskell's type system[7][1] to implement the API of a web enabled server. Our application uses this specification to generate statically-checked callback URLs and to uniformly map those callback URLs to functionality in the application. The API description can then also be used to generate documentation for the server's API. The benefit of this approach is that we are able to separate concerns about processing user requests from the core logic of our application in a statically typed manner. Because of this, we detect potential errors at compile time, ensuring that the server interface is well defined, documented, and that callbacks to the server match the interface. This is especially valuable in the development stages of our server when the API was still in flux.

We illustrate the basic idea with a running example. The web application is a document server, and the client is a web browser. The client needs to access a particular version of a document stored on the server. In this example, the client might request the document by using a URL of the following form:

```
/view?title=MyDoc&rev=10
```

The path of the URL contains the name of the method on the server that should be invoked (in this case `view`), and the query string (the text after the question mark) contains the method arguments. The arguments are passed by using the standard form encoding for URLs [1]: different arguments are separated by `&`, and the name and value of the arguments are separated by `=`. In this example, we have two arguments named "title" and "rev" with values "MyDoc" and "10" respectively. In the server, this interface is implemented using the following Haskell code:

```
view :: String -> Maybe Int -> Response
view title rev = (...)

viewSig =  Method
  (  (Arg "title" :: Arg Req String)
  :> (Arg "rev"   :: Arg Opt Int)
  )
  "view" "View a document"

(handlers, docs) = unzip
  [ viewSig --> view
  , (...)
  ]
```

The actual handler is `view`, which is an ordinary Haskell function that takes two arguments: a `String` and an optional `Int`. The external API for the method is specified in `viewSig`: here we specify the name of the method, what it does, and the arguments it expects. Finally, we use the operator `(-->)` to link the specifications with their handlers.

Paper overview The rest of the paper is organized as follows: In Section 2 we describe our implementation; in Section 3, we show how this framework may be extended to support additional features; in Section 4, we discuss some of the Haskell extensions that make this interface possible and conclude.

[1] With commonly used extensions to be discussed in Section 4.

2. Implementation

In this section, we present our implementation of the web function framework. We start by describing how to represent individual arguments, then we proceed with machinery for working with lists of arguments, and finally we put it all together to describe the methods of an API.

2.1 Individual Arguments

First we need to specify how to represent parameter values as strings. These are the external representations of the values and are used when we parse and construct URLs corresponding to API calls. The `ArgRep` class provides methods to support serialization and documentation:

```
class ArgRep a where
  show_arg  :: a -> ShowS
  read_arg  :: String -> Maybe a
  show_type :: f a -> String
```

The functions `show_arg` and `read_arg` convert argument values to and from their string representation respectively. The function `show_type` returns a human readable string for the type of the argument, which we use for generating documentation. The polymorphic `f` in `show_type` guarantees that `show_type` will only be able to generically report the type of its argument.

Argument Signatures An important part of the signature for a method in the API is its argument list. Each entry in the list is an *argument signature* that specifies the name of the argument, its type, and if it is a required or an optional argument. Argument signatures are specified with the following datatype:

```
data Arg a b = Arg String
data Req
data Opt
```

The `String` value in the type `Arg` is the name of the argument. We use phantom types[5] to specify the other properties of the arguments. The first parameter, `a`, specifies if the parameter is required (written with the type `Req`) or optional (written with the type `Opt`). The second parameter, `b`, specifies the type of the argument value. For example, this is how we would define the signature for an optional `String` argument called "title":

```
aTitle = Arg "title" :: Arg Opt String
```

Manipulating Signatures Next, we need a way to associate the components of a URL with the parameters of a method. We process URLs by using a separate library, which presents the query string as a map associating argument names with the string representation of their values:

```
type ArgMap = [(String,String)]
```

This abstraction is also useful when the method arguments come from a source different then the query string of a URL (e.g., parameters may be extracted from the fields of a posted HTML form).

We process optional and required arguments in a different way. For optional arguments we use the `Maybe` type to indicate if the user provided a value for the parameter, while if a required argument was missing we would signal an error. By using the type class we are able to vary the mapping between argument type and result type using the based on the lookup strategy.

```
class ArgRep a => Strategy r a c | r a -> c where
  arg_in  :: Arg r a -> ArgMap -> Either String c
  arg_out :: Arg r a -> c -> ArgMap
  arg_doc :: Arg r a -> ShowS
```

The type arguments for the class are as follows: `r` specifies if an argument is required or optional, `a` is the type of the argument value, and `c` is the type of the result that is expected by the handler for the particular request. For required arguments, `a` and `c` are always the same, but for optional arguments `c` will be `Maybe a`:

```
instance ArgRep t => Strategy Req t t ...
instance ArgRep t => Strategy Opt t (Maybe t) ...
```

The method `arg_in` is used to locate the value of an argument within an argument map. The method `arg_out` is used when we construct callback URLs, and produces the part of the query string that will be used to encode the given argument. Finally, the method `arg_doc` generates documentation for the argument, including its name, type, and if it is required or optional.

2.2 Lists of Arguments

It is common for methods to have multiple arguments, so we need a way to represent lists of arguments. We cannot use ordinary lists because different arguments may be of different types, and this is reflected in the argument signatures. Instead, we use a tuple-like cons cell:

```
data a :> as = a :> as
infixr 5 :>
```

This operator introduces types that are essentially tuples but the infix notation makes signature specifications significantly easier to read and write because we avoid needing nested parentheses. Continuing our example, we define the argument list for the example from Section 1:

```
view_args :: Arg Req String :> Arg Opt Int
view_args  = Arg "title"    :> Arg "rev"
```

In the rest of this section we shall use the list-like structure of the `a :> b` type to define URL generation, documentation generation, and handler application as a special kind of fold. In each case we use a type class to define the recursion as we are dealing with heterogeneous lists [8, 4].

Documenting Argument Lists The `ArgsDoc` class provides a method for folding an argument list into a human-readable string. We use these strings to provide documentation for our application's API and also to pin-point where an error might have occurred when parsing arguments. By producing the documentation directly from the API description values, we are more likely to keep it up-to-date than if we were to hide that information away in comments or rely on an external tool to produce documentation.

```
class ArgsDoc a where
  args_doc :: a -> ShowS

instance ArgsDoc () where ...

instance (Strategy x a k, ArgsDoc args) =>
  ArgsDoc (Arg x a :> args) where ...

instance Strategy r a c =>
  ArgsDoc (Arg r a) where ...
```

The first instance is the base case which is used for methods that have no arguments. The inductive step is in the second instance, which uses `arg_doc` to document the head of the argument list, and then it appends this to the recursively generated documentation for the rest of the arguments. Note that the recursive call will use `args_doc` at a different type, which is why we need a type class to define such functions. Technically, these two instances are sufficient to achieve everything that we need. However, for convenience, we add the third instance which provides another base case. This last instance enables us to work with argument lists, like `view_args` that do not have `()` as their last component.

Callback URLs The `Build` class defines a fold from argument lists to functions that generate `URL`s. Because we expose the types of the arguments in our `(:>)` type, we are able to create functions that take exactly the arguments needed to generate the `URL`. This functionality is very useful when we work with dynamically generated web pages that contain callbacks to the server (e.g., a link that requests the previous version of a document). By using our technique, we know that such web pages will not contain malformed server callbacks.

```
class Build a b | a -> b where
  build :: a -> URL -> b

instance Build () URL where ...

instance (Strategy r a c, Build args f) =>
  Build (Arg r a :> args) (c -> f) where ...

instance Strategy r a c =>
  Build (Arg r a) (c -> URL) where ...
```

These instances follow a similar pattern to the ones for generating documentation but we use `arg_out` to compute what fields to add the URL. The main structural difference from the documentation case is that here we use a two parameter type class with a functional dependency [6], which computes the type of the resulting URL generating function.

As an example, consider what happens when we apply `build` to `view_args`: the result is a function of the appropriate type ensuring that values for all of the defined arguments are added to the URL:

```
view_url_args :: URL -> String -> Maybe Int -> URL
view_url_args = build view_args
```

Request Handlers The `Apply` class defines a way to extract arguments from a URL (in the form of `ArgMap`) and then applies a function to them, as specified by a list of argument signatures. It completes by either returning the result of the function, or an error message explaining why it was not able to extract an argument from the `ArgMap`.

```
class Apply args func res
  | args res -> func, args func -> res where
  apply :: args -> func -> ArgMap -> Either String res

instance Apply () res res where ...

instance (Strategy r a c, Apply args f res) =>
  Apply (Arg r a :> args) (c -> f) res where ...

instance (Strategy r a c) =>
  Apply (Arg r a) (c -> res) res where ...
```

Again, the instances have a similar form to the previous two definitions. This time, we have an extra argument for the final result of applying the function. The two functional dependencies state that the argument list paired with the function uniquely determines the result, and vice versa—the result determines the function.

As an example of how `apply` works, consider what happens when we use it with the `view_args` list of arguments:

```
view_handler :: (String -> Maybe Int -> c)
             -> ArgMap -> Either String c
view_handler = apply view_args
```

2.3 Putting It All Together

The API of a server is specified by providing a number of *method signatures*. For each method we need to provide a name, an argument list, and a description of its functionality. Because different methods have arguments of different types, we parametrize methods by their argument lists.

```
data Method args = Method
 { meth_args  :: args
 , meth_name  :: String
 , meth_doc   :: String
 }
```

By combining the information in `Method` values, with the functionality for processing arguments from the previous section, we obtain a number of useful functions for the API. For example, we can define a function that generates the documentation for a method:

```
methodDoc :: ArgsDoc args
          => Method args -> String
```

The implementation of this function uses the name and the description of a method from the method signature, and it also uses `args_doc` to document the method arguments. For example, the documentation for the method `viewSig` from Section 1 is as follows:

```
view: View a document
  title :: String
  rev   :: Int (optional)
```

It is just as easy to define a function that generates URL callbacks for a given method:

```
methodURL :: Build args function
          => Method args -> function
```

This function defines a base URL that has the method name in the path, and then uses `build` to create a correctly typed function that will add the encoded arguments to the URL query string. For example, if we were to apply `methodURL` to `viewSig`, then we will get a function of the following type:

```
viewURL :: [Char] -> Maybe Int -> URL
viewURL = methodURL viewSig
```

In a similar fashion we can associate method signatures with the functions that implement the method (we call such functions the *method handlers*):

```
methodHandler
  :: Apply args handler result
  => Method args -> handler
  -> String -> ArgMap -> Maybe (Either String result)
methodHandler m h path arg_map =
  do guard (meth_name m == path)
     return (apply (meth_args m) h arg_map)
```

As with `methodURL`, we use different types of handlers for methods with different arguments, which is why the function is polymorphic in `handler`. The `String` and `ArgMap` parameters correspond to the path and query string of a URL request. The result of `methodHandler` can be interpreted as follows: `Nothing` indicates that the request cannot be handled by this method; `Just errs` will be returned if the request matched the method name but we encountered errors while processing the method arguments; if there were no errors with arguments, then we execute the handler and return `Just result`.

Often it is convenient to associate a method signature with its handler, and generate the documentation for that method at the same time. To make code like that more readable, we define a simple infix operator:

```
sig --> handler =
  (methodHandler sig handler, methodDoc sig)
```

By using this operator, we can concisely specify the server's API in a single place, by using a pattern like this:

```
(handlers,docs) = unzip
  [ sig1 --> handler1
  , sig2 --> handler2
  ...
  ]
```

Here `sig1`, `sig2`, etc are the method signatures for the various API methods, and `handler1`, `handler2`, etc. are the server functions that should be invoked to handle the corresponding requests. The value `docs` contains documentation for each of the methods, while `handlers` is a list of functions that will attempt to handle a request. We can put all the handlers into a single request handler by using the function `runAPI`:

```
runAPI
  :: [String -> ArgMap -> Maybe (Either String a)]
  -> String -> ArgMap
  -> Maybe (Either String a)
runAPI hs p as = msum [ h p as | h <- hs ]
```

This function picks the first handler that accepts the given requests. Its result can be interpreted as follows: `Nothing` means that no request handler applies, in which case we may want to redirect to a default handler or signal an error and display the list of available methods; `Just errs` indicates that we found a handler but there was a problem with decoding and validating its arguments; finally, if there were no problems, then we return `Just result`.

3. Other Features

Our framework lends itself to various extensions, many of which we use in our applications already. These include: argument documentation, argument validation, multi-valued arguments, and anonymous arguments. In this section, we briefly describe these ideas.

Argument documentation and validation can be implemented easily by modifying the type `Arg` like this:

```
data Arg' a b = Arg'
  { arg_name        :: String
  , arg_doc_string :: String
  , arg_valid       :: a -> [String]
  }
```

The field `arg_doc_string` contains documentation that is specific to this argument, while the field `arg_valid` contains a validating function that can examine values and return a list of problems with them. Argument validation can be performed naturally after we parse an argument. If we detect validation errors, we can return an appropriate response to the user and provide an opportunity to make corrections. It may also be feasible to perform validation on whole lists of arguments, which may be achieved by modifying the implementation of argument lists.

In web applications it is common to present lists of information to the user. Often it is necessary to associate one or more form elements to each of the elements in the list. We can achieve this by adding a new constructor, `Many`, to supplement `Req` and `Opt`. To specify the behavior of such arguments we add a new instance to the class `Strategy` that collects all arguments with a common prefix (the name of the argument) and constructs a map value that associates the name suffixes with the corresponding values.

Often it can be visually appealing to embed arguments to methods directly in the path of a URL. For example, in the following URL `http://example.com/view/98`, the number '98' might be a parameter to the 'view' method. By adding an additional type to the argument list we can define instances to read and write such anonymous arguments from the path.

4. Summary and Discussion

Most web applications are written in a way where the user-interface is defined in an ad-hoc manner. Since there is no formal place that the application's functionality is defined, it can be difficult to verify that documentation and calls to that functionality stay up-to-date as development proceeds. We implemented a Haskell library that attempts to address these common concerns in web-development by allowing the compiler to verify that the web-application generates valid URLs to well specified and automatically documenting functionality.

Our technique is not limited to URL-based web-applications. It can be applied in other RPC-based situations where the client of an application is presented with a less typed interface than what is available within the sever. Examples of such situations include applications that communicate via JSON RPC [2] and command-line interfaces.

Haskell Extensions In developing the request dispatch framework, we used a number of extensions to Haskell 98 [7] that are implemented in the GHC compiler[3]. Some of the more mundane extensions are support for datatypes with no constructors and infix type operators. The first of these is quite common when using the type-level programming techniques that we used in our implementation because these empty types are essentially the constructors of a user-defined *kind*. Adding support for user-defined kinds would remove the need for empty data declarations in applications like ours. The second extension, type level operators, is just syntactic sugar but it made the method signatures of our API much nicer to look at, so we consider it to be quite useful.

We also used a number of extensions to Haskell's class system, namely multi-parameter type classes with functional dependencies and flexible instances[2]. All of these are essential for writing programs in the style that we presented in this paper.

Our Experience The statically checked server API proved useful on a number of occasions when we made changes to the API but forgot to update all the necessary parts of our program. Such mistakes were detected at compile time and were easy to rectify. A common critique of libraries that use sophisticated type level tricks is that type errors become difficult to understand. For this particular application this has not been a big problem, because the top level server API is monomorphic. Overall, our experience in using Haskell for this application was quite positive.

References

[1] HTML 4.01 Specification. `http://www.w3.org/TR/html4/`.

[2] JSON-RPC. `http://json-rpc.org/`.

[3] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[4] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[5] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[6] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, Berlin, Germany, March 2000.

[7] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[8] Thomas Hallgren. Fun with Functional Dependencies. In *Proceedings of the Joint CS/CE Winter Meeting*, Varberg, Sweden, January 2001.

[2] The current version of GHC, 6.8.2, also requires an option called *undecidable instances* although solving constraints with the concrete instances that we used is decidable.