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.

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 polymor-

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