Resource-Based Web Applications*

Sebastian Fischer
sebf@informatik.uni-kiel.de

Christian-Albrechts-University of Kiel, Germany

Abstract

We present an approach to write web applications in the programming languages Haskell [8] and Curry [6]. The web applications we propose are directly based on the Hypertext Transfer Protocol (HTTP) – no additional protocol on top of HTTP is necessary. Although the main ideas are not new and already applied using other programming languages, we believe that features present in modern declarative languages – especially algebraic datatypes – serve well to represent resources on the web.

1 TRANSFER OF RESOURCE REPRESENTATIONS

A *resource* is any piece of information named by a Uniform Resource Identifier (URI). A web application provides access to internally stored information over the Internet or accesses such information provided by other web applications. Usually, (a representation of) this information is transferred using the Hypertext Transfer Protocol (HTTP) [3] – possibly employing some protocol on top of it. Note that we do not primarily consider applications running inside a web browser like [4, 7, 9].

HTTP provides different methods to access resources from which the most important are the GET and the POST method. GET is used to *retrieve* some information, e.g., web browsers use GET to ask for the content of web pages. POST is used to *send* information to be processed by a server, e.g., entries in a web form can be sent to the server using POST. While processing the request, the server can update its internally stored information and usually sends back a result of this process.

Web applications differ in their use of these two HTTP methods: RPC-style web applications are accessible through a single URI and communication is performed using the POST method. A representation of a procedure call is transferred to the server, which returns a representation of the result of this call as answer to the POST request. Note that both retrieval and update operations are performed using the POST method in RPC-style web applications and the interface of such applications is determined by the procedure calls that are understood by the server.

Fielding [2] proposes another architectural style for web applications: He proposes to use the POST method only to update the internal state of the server and to use the GET method to purely retrieve information. Consequently, everything that should be available for retrieval by clients needs to be identified by an own URI. This is a big difference to RPC-style web applications that are usually accessible

*This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-1.

through a single URI. In the architectural style proposed by Fielding, a client sends an HTTP GET request to an URI to retrieve a representation of the resource identified by this URI corresponding to the current state of the server. To change this state, the client sends a POST request to the URI of the resource he wants to update. Fielding calls this architectural style Representational State Transfer (REST).

The interface of a RESTful web application is given by the methods of HTTP and the URIs identifying the applications resources. In this work, we show how the interface given by the most important HTTP methods can be made available to Haskell or Curry programmers, hiding the details of HTTP communication.

2 A RESOURCE API

We assume that the reader is familiar with the syntax of the programming language Haskell [8] or Curry [6]. In our approach, the programmer accesses resources as values of type `Resource a b c` which provide access to local or remote resources. For this purpose the operations

```
get :: Resource a b c -> IO a
update :: Resource a b c -> b -> IO c
```

are provided. The type variable `a` denotes the type of the result of a resource retrieval. The type variables `b` and `c` denote the argument and return type of the operation updating a resource.

The operations shown above model the GET and POST methods of HTTP: The `get` operation is an IO action that takes a resource and returns the current state of this resource which is always of type `a`. To perform `get` on a remote resource, an HTTP GET request is sent to the corresponding URI and the response is converted into a value of type `a`. The `update` operation is an IO action that takes a resource along with a value of type `b` and returns a value of type `c` as result. To perform an `update` on a remote resource, a representation of the value of type `b` is sent to the corresponding URI using HTTP POST and the response to this request is converted into a value of type `c`. Access to a remote resource can be obtained by the function

```
remoteResource :: URI -> Resource a b c
```

This signature is not completely honest, because in addition to the URI the programmer needs to specify how the transferred data has to be converted. For the moment, we skip the details of this conversion.

2.1 An Example: Access to a News Provider

In our framework we can model both RPC-style and RESTful web applications: The former would ignore the `get` operation and use `update` to send representations of procedure calls as values of type `b` to the server. As an example for the latter, we consider a news provider:

To model an interface to news as algebraic datatypes, we define two data structures. The first represents a reference to the second, and can be used in summaries. Note that these datatypes reflect the data transferred over the web. They are independent of the internal representation of news chosen by the server. Internally, the server does not need to distinguish news references from news items and can avoid redundant storage of headlines and dates.

```
data NewsRef = NewsRef Headline Date URI
data NewsItem = NewsItem Headline Date Text [Comment]
```

If the news application provides a list of all headlines at the URI `www.example.net/news`, we can create a remote resource representing this list

```
newsList :: Resource [NewsRef] NewsItem NewsRef
newsList = remoteResource "www.example.net/news"
```

and retrieve a list of the currently available headlines by `get newsList` which could return something similar to[1]

```
[NewsRef
  "TFP extended abstract submission tomorrow"
  "2006/2/16"
  "www.example.net/news/tfp-deadline"]
```

The URI given as third argument of `NewsRef` is a link to the news item corresponding to the headline. We can also access this news item using a resource

```
tfpDeadline :: Resource NewsItem Comment NewsItem
tfpDeadline = remoteResource
  "www.example.net/news/tfp-deadline"
```

and retrieve the item by `get tfpDeadline` which could return

```
NewsItem
  "TFP extended abstract submission tomorrow"
  "2006/2/16"
  "The deadline for extended abstracts ..."
  []
```

To add a comment to this news item, we can call

```
update tfpDeadline "this is quite soon!"
```

and would get

```
NewsItem
  "TFP extended abstract submission tomorrow"
  "2006/2/16"
  "The deadline for extended abstracts ..."
  ["this is quite soon!"]
```

[1]For simplicity, we encode headlines, dates and URIs as strings.

as the result of this call. If we now perform the action `get tfpDeadline` again, we get the same result.

Consider the three arguments of the type constructor `Resource` given in the definition of `tfpDeadline`: The first argument is `NewsItem` which means that if we call `get` on this resource we get a result of type `NewsItem`. The second and third arguments are `Comment` and `NewsItem`, respectively. So if we call `update` on this resource, we have to provide a value of type `Comment` and get a result of type `NewsItem`.

Now recall the corresponding types in the definition of `newsList`: We called `get` on this resource and got a list of `NewsRefs` which corresponds to the first type argument `[NewsRef]`. The other arguments are `NewsItem` and `NewsRef`, so we can call `update` on this resource with corresponding values:

```
update newsList
 (NewsItem
   "TFP deadline for extended abstracts expired"
   "2006/2/18"
   "The deadline for extended abstracts ..."
   [])
```

As a response to this call, the server could add this news item to its internal news database and return a news reference with a newly created URI pointing to the new item:

```
NewsRef
  "TFP deadline for extended abstracts expired"
  "2006/2/18"
  "www.example.net/news/tfp-deadl-exp"
```

In this example the different resources representing a list of news headlines and complete news items are connected by URIs. This is not an accident but typical for RESTful web applications. Just like ordinary web pages, resources can be interconnected by links, because everything that can be retrieved by clients is identified by a unique URI. If we want to take this idea one step further as we did in our example, we could include references to related news items in every news item.

The news provider may want to restrict the access to its internal news database. For example she could require an authentication to add comments to news items or create new items in the internal database. HTTP provides methods for authenticated communication and we integrate authenticated HTTP communication in our approach by the function:

```
authResource :: String->String->URI->Resource a b c
```

The two additional arguments of type `String` specify the user name and password that identify the client.

3 DATA AND ITS REPRESENTATION

In Section 2 we omitted the details of data conversion for the transfer over the Internet. Unfortunately, Haskell (as well as Curry) values cannot be transferred directly by HTTP but have to be converted. The payload of HTTP messages can be of arbitrary type, however, usually a textual representation is employed. So we need to be able to convert program data into a textual form. Both Haskell and Curry provide functions to convert values into strings and vice versa, so we can use these functions to convert program data for HTTP transfer.

However, the direct representation of algebraic datatypes is language dependent. If we want to share information with as much as possible other participants on the Internet, we need to use an independent format. The standard format for data representation on the Internet is XML and this is where modern declarative languages expose an advantage: Algebraic datatypes are tree-like structures that can be represented easily in XML [10]. Although there are also XML language bindings for, e.g., object oriented languages, values of an algebraic datatype are much closer to XML terms than objects.

Another language independent data format is the Javascript Object Notation (JSON)[2] which is sometimes preferred to XML because of its brevity. We support both XML and JSON in our approach and we want the user of our tool to be able to support other formats as well so we do not hide data conversion completely. The user should also be able to support more than one format in a single web application by allowing the clients to choose the format according to their needs. We use HTTP content negotiation for this purpose, where the client sends a list of content types that he is willing to accept in the header of an HTTP request.

3.1 Content-Type-based Conversion

The function `remoteResource` mentioned in Section 2 has additional arguments that control data conversion:

```
remoteResource :: Conv a -> Conv b -> Conv c -> URI
               -> Resource a b c
```

The type `Conv a` is defined as

```
type Conv a = [(ContentType, ReadShow a)]
type ReadShow a = (String -> Maybe a, a -> String)
data ContentType
  = Plain
  | Xml
  | Json
  ...
  | CT String
```

[2]http://www.json.org/

Values of type `ContentType` represent content types that are specified in the header of HTTP messages. To access a remote resource, the converters are used as follows:

- When `get` is called on a resource, an HTTP GET request is sent to the corresponding URI and the response is parsed with the read function that is associated with the content type of the response in the list of type `Conv a`. The read function returns a value of type `Maybe a` and should indicate malformed data by returning `Nothing`.
- When `update` is called on a resource and a value of type `b`, this value is converted using the first show function in the list of type `Conv b` and sent to the corresponding URI using HTTP POST. The result of this request is parsed with the read function that is associated with the content type of the response in the list of type `Conv c`. If the POST request fails, the other show functions are tried successively.

We provide a function `plainConv`[3] that converts values of arbitrary instances of the type classes `Read`[4] and `Show`[5]:

```
plainConv :: (Read a, Show a) => Conv a
plainConv = [(Plain, (safeRead, show))]

safeRead :: Read a => String -> Maybe a
safeRead s = case (reads s) of
               [(x,"")] -> Just x
               _ -> Nothing
```

3.2 Type-based Construction of Converters

The implementation of `plainConv` is very simple. However, for applications that communicate with other applications - possibly written in different programming languages - over the Internet, we need to convert data into a language independent format. Implementing these converters by hand is a complex, error prone and usually tedious task. Hence, we provide a framework to concisely construct such converters using a run-time type specification of type `ConvSpec a`. We provide primitive specification functions

```
cInt    :: String -> ConvSpec Int
cString :: String -> ConvSpec String
cBool   :: String -> String -> ConvSpec Bool
```

to construct converters for values of type `Int`, `String` and `Bool`. The arguments of type `String` specify labels that are used for the conversion. For values of type

[3]Here we represent content types as strings for simplicity.

[4]Note that `(read . show)` should be the identity function.

[5]Since Curry lacks type classes, we use the functions `readsQTerm` and `showQTerm` instead.

`Bool` one label for each `True` and `False` is specified. More complex converters can be build by combinators like

```
cList    :: String -> ConvSpec a -> ConvSpec [a]
cMaybe   :: String -> String -> ConvSpec a
         -> ConvSpec (Maybe a)
cPair    :: String -> ConvSpec a -> ConvSpec b
         -> ConvSpec (a,b)
cTriple :: String
         -> ConvSpec a -> ConvSpec b -> ConvSpec c
         -> ConvSpec (a,b,c)
...
cEither :: String -> String
         -> ConvSpec a -> ConvSpec b
         -> ConvSpec (Either a b)
```

Since all datatypes can be mapped to values of types covered by the presented combinators, these are sufficient to build converters for arbitrary datatypes. We provide a function

```
cAdapt   :: (a -> b) -> (b -> a) -> ConvSpec a
         -> ConvSpec b
```

to construct a converter for a type that is not covered by the presented combinators. The first two arguments convert between this datatype and another datatype with converter specification of type `ConvSpec a`.

We provide implementations of these combinators in Curry to construct converters for XML, JSON and both XML and JSON at the same time. For example, to support XML conversion we define

```
type ConvSpec a = (XmlExp -> a,a -> XmlExp)
```

and `cPair` could be defined as

```
cPair tag (rda,sha) (rdb,shb) = (rd,sh)
 where
  rd (XElem [] t [a,b]) | t==tag = (rda a,rdb b)
  sh (a,b) = XElem [] tag [sha a,shb b]
```

In Curry, we can employ the concept of function patterns [1] and provide more general combinators to construct converters for arbitrary record types directly. Function patterns extend the notion of patterns by applications of defined operation symbols.

A generalization of `cPair` can be defined using function patterns as[6]:

```
cPairCons   :: (a -> b -> c) -> String
            -> ConvSpec a -> ConvSpec b
            -> ConvSpec c
cPairCons cons tag (rda,sha) (rdb,shb) = (rd,sh)
 where
  cf a b = cons a b
  rd (xml t [a,b]) | t==tag = cons (rda a) (rdb b)
  sh (cf a b) = xml tag [sha a,shb b]
```

Another advantage of Curry over Haskell is that we can employ nondeterminism to specify converters for datatypes with multiple constructors. For example a converter specification for binary trees can be defined as:

```
data Tree = Leaf Int | Branch Tree Tree

cTree :: ConvSpec Tree
cTree = cCons Leaf "leaf" (cInt "value")
cTree = cPairCons Branch "branch" cTree cTree
```

In Curry multiple rules are not applied in a top-down approach. Instead every matching rule is applied nondeterministically.

We provide a function

```
conv :: ConvSpec a -> Conv a
```

that generates a converter from a specification. The programmer can decide to create different kinds of converters by importing different modules. The modules are designed such that only import declarations and no other code has to be changed in order to change the generated converters.

As an example application of the provided combinators, consider converter specifications for the news datatypes presented in Section 2.1:

```
cNewsRef :: ConvSpec NewsRef
cNewsRef
  = cTripleCons NewsRef "ref"
      (cString "headline")
      (cString "date")
      (cString "uri")

cNewsRefs :: ConvSpec [NewsRef]
cNewsRefs = cList "refs" cNewsRef
```

[6] `xml` is defined as `XElem []`, i.e., it constructs an XML element with empty attribute list.

```
cNewsItem :: ConvSpec NewsItem
cNewsItem
  = c4Cons NewsItem "item"
      (cString "headline")
      (cString "date")
      (cString "text")
      (cList "comments" cComment)

cComment :: ConvSpec Comment
cComment = cString "comment"
```

This is everything we need to define to get conversion functions for both the XML and JSON format!

The presented combinators construct complex converter functions from concise definitions. Thus, they eliminate a source of subtle errors and release the programmer from the burden to write such converters by hand. The programmer only has to give a specification that resembles a type declaration augmented with labels, that identify components of complex datatypes.

4 SERVER APPLICATIONS

In the previous sections we always considered the access of remote resources. In this section we describe how to provide local resources through a web-based interface. An interface to a local resource is created by the function

```
localResource :: IO a -> (b -> IO c)
                -> Resource a b c
```

that takes two IO actions performing the `get` and `update` operations of the created resource. The resource operations `get` and `update` provide a uniform interface to both local resources and remote resources available via HTTP. The programmer can access a remote resource in the same way she accesses local resources and does not need to deal with the details of communication.

To provide a web-based interface to a (not necessarily) local resource, the programmer can use the functions:

```
provide :: ConvSpec a -> ConvSpec b -> ConvSpec c
        -> Resource a b c -> Handler

cgi     :: (Path -> IO (Maybe Handler)) -> IO ()
```

The function `provide` takes converter specifications for the types `a`, `b` and `c`, a resource of type `Resource a b c` and returns a value of the abstract type `Handler` that handles GET and POST requests. The function `cgi` takes an IO-action that computes an optional handler corresponding to path information and returns an IO-action that acts as a CGI program. Both functions will be discussed in more detail in Section 5 after we reconsider the news example.

4.1 Example Continued: The News Provider

In Section 2.1 we introduced a client to a news provider. In this section we show how the provider itself can be implemented using our framework. Suppose we have a local news database that supports the following operations:

```
getNewsList :: IO [NewsRef]
getNewsItem :: Path -> IO NewsItem
addNewsItem :: NewsItem -> IO Path
addNewsComment :: Path -> Comment -> IO ()
```

With these operations we can implement the news provider introduced in Section 2.1.

We define a function `newsProvider` that dispatches requests to appropriate handler functions:

```
newsProvider :: Path -> IO (Maybe Handler)
newsProvider path
  | path == ""
   = return (Just
       (provide cNewsRefs cNewsItem cNewsRef
         (localResource getNewsList addItem)))
  | otherwise
   = refs <- getNewsRefs
     if readInt path `elem` refs
      then return (Just
            (provide cNewsItem cComment cNewsItem
              (localResource
                (getNewsItem path)
                (addComment path))
      else return Nothing
```

We need to provide functions `addItem` and `addComment` based on the database operations given above:

```
addItem :: NewsItem -> IO NewsRef
addItem item@(NewsItem headline date _ _) = do
  path <- addNewsItem item
  return (NewsRef headline date (myLocation++path))
```

The string `myLocation` represents the URI pointing to the CGI program serving the requests. The function `addComment` returns the updated news item:

```
addComment :: Path -> Comment -> IO NewsItem
addComment path comment = do
  addNewsComment path comment
  getNewsItem path
```

We have shown the complete implementation of a news server that provides access to an internal news database via an XML-, JSON- and plain-text-based interface. Clients communicate with the database via HTTP GET and POST requests which are served by a CGI program.

5 IMPLEMENTATION

In this section we discuss the library functions introduced above in more detail. We discuss a representation for HTTP messages, the implementation of the GET and POST handler functions and explain what errors are handled automatically within our framework.

5.1 Hypertext Transfer Protocol

Since communication is performed using HTTP, we model HTTP messages as algebraic datatype `Http`:

```
data Http = Http Method [Header] String

data Method
  = Get Path
  | Post Path
  | Respond Status
  ...
  | Method String

data Status
  = Ok
  | BadRequest
  | NotFound
  | MethodNotAllowed
  | NotAcceptable
  ...
  | Status Int String

data Header
  = Accept [ContentType]
  | ContentType ContentType
  | ContentLength Int
  ...
  | Header String String
```

An HTTP message consists of a *method*, a list of *header fields* and a *message body*. In client messages the method specifies the type of the request. For example, GET and POST are HTTP methods that both refer to a (local) path of a resource.

Server messages contain an *initial response line* instead of a method and indicate the response status with a *status code*. The most important response status is *Ok*, which indicates a successful response. The other status codes shown above all indicate errors. Some of them will be discussed in the following section. Briefly, the status *Bad Request* indicates a malformed client message. *Not Found* is sent if the specified resource is not available. The server sends *Method Not Allowed* if he does not support the method of the client message. *Not Acceptable* indicates that the client requested a content type, that is not available.

Every HTTP message can contain an arbitrary number of header fields. For example, the client can use the *Accept* header field to indicate, what content types he accepts in the response message. The server specifies the content type of the response in the *Content-Type* header field and the size of the message body in the *Content-Length* header field. We provide a function

```
showHttp :: Http -> String
```

to transform a value of type `Http` into its textual representation and a function

```
readHttp :: String -> Http
```

to read a value of type `Http` from a string. Since usually the size of the message body is not known in advance but specified in the HTTP message itself, we provide an IO-action

```
hGetHttp :: Handle -> IO Http
```

that takes a (readable) IO-handle and returns an HTTP message read from this handle.

5.2 Common Gateway Interface

The easiest way to build a server application for a resource is to create an executable CGI program. Such a program takes inputs from *stdin* and the environment and sends its output to *stdout*. The details of this communication are defined in the Common Gateway Interface and are completely hidden by our library. It would also be possible to create an application that directly connects to a port and acts as a server. However, we did not consider this possibility, because a server is more complex to use as well as to implement than a CGI program that can be used together with existing web server software. A web application has to answer GET and POST requests. In a first step, we provide functions that answer such requests using a single resource:

```
type GetHandler   = [ContentType] -> IO Http
type PostHandler  = ContentType -> String
                    -> [ContentType] -> IO Http
```

```
handleGet   :: Conv a -> Resource a b c
            -> GetHandler
handlePost  :: Conv b -> Conv c -> Resource a b c
            -> PostHandler
```

The presented handler functions proceed as follows:

- `handleGet` performs the operation `get` on the supplied resource and the result is converted into a textual representation using a show function in the list of type `Conv a`. This representation is wrapped in an HTTP message that can be sent back to the client. If the request accepts only specific content types for the response an appropriate show function is selected.

- `handlePost` parses the given string with the read function in the list of type `Conv b` that is determined by the given content type. The resulting value is supplied to the operation `update` along with the resource and the result of this operation is converted into an appropriate textual representation by a show function in the list of type `Conv c`. Again, this representation is wrapped in an HTTP message to be sent back to the client.

We provide a shortcut for creating both a GET and a POST handler for a resource at the same time:

```
type Handler = (GetHandler, PostHandler)

handle  :: Conv a -> Conv b -> Conv c
        -> Resource a b c -> Handler
```

For convenience, we also provide a function that computes a `Handler` from `ConvSpec`'s instead of `Conv`'s:

```
provide  :: ConvSpec a -> ConvSpec b -> ConvSpec c
         -> Resource a b c -> Handler
```

Since RESTful web applications provide URIs for everything that can be retrieved by clients, one resource is usually not enough to model the application. In fact, every path extending[7] the location of the created CGI program points to a new resource. Unlike [7], we explicitly handle multiple URIs with a single CGI program. We provide a function `cgi` that computes an IO action that can be compiled into a CGI program from a given IO-action that maps `Paths` to GET and POST handler functions:

```
cgi :: (Path -> IO (Maybe Handler)) -> IO ()
```

A request to the created CGI program is dispatched to the appropriate handler functions by the given mapping which is applied to the path that identifies the requested resource.

[7]Such paths are provided in the environment variable PATH_INFO available to CGI programs.

5.3 Error Handling

There are several situations, where the server should deliver an error message in response to a clients request. For example, if a resource specified by the client is not available on the server, the server should indicate this with an *Not Found* status in the response message. This error is the reason for the `Maybe` type in the declaration of the function `cgi` given above: If the supplied IO-action returns `Nothing` for a given path, the CGI program generates a *Not Found* error message.

The functions `handle` and `provide` generate both GET and POST handlers for a given resource. However, if the HTTP method of the request is neither GET nor POST the computed handler generates a *Method Not Allowed* error message. We also provide functions

```
handleGetOnly    :: Conv a
                 -> Resource a b c -> Handler
provideGetOnly   :: ConvSpec a
                 -> Resource a b c -> Handler
```

and

```
handlePostOnly   :: Conv b -> Conv c
                 -> Resource a b c -> Handler
providePostOnly  :: ConvSpec b -> ConvSpec c
                 -> Resource a b c -> Handler
```

These functions generate a handler for only one HTTP method and generate a *Method Not Allowed* error message for other methods.

The type `Conv a` that specifies data conversion was defined in Section 3.1 as

```
type Conv a = [(ContentType, ReadShow a)]
```

This list contains one pair for every content type supported by the application. If

1. the client uses the *Accept* header field to specify which content type she accepts in the response message and

2. none of the accepted content types is supported by the application

the CGI program generates a *Not Acceptable* error message as response to the request. The type `ReadShow a` was defined as

```
type ReadShow a = (String -> Maybe a, a -> String)
```

The read function returns a value of type `Maybe a` and should return `Nothing` if the given string cannot be parsed. When handling a POST request, the CGI program uses such a read function to parse the message body of the request. If the result is `Nothing` it generates a *Bad Request* error message as response.

6 RELATED AND FUTURE WORK

Hanus [4], Meijer [7] and Thiemann [9] provide frameworks to build server side web applications based on HTML forms that run inside a web browser. Although [7] abstracts from network communication, it uses strings as references to input fields, which is a source of possible errors. [4, 9] abstract also from such references, which not only prevents typing-errors but also enables the programmer to compose different web forms without the risk of name clashes. Recently, Hanus [5] presented an approach to concisely describe type-based web user interfaces (WUIs), based on [4].

Our approach differs from the mentioned approaches because we do not aim at web based *user* interfaces through HTML forms but provide a framework for web based interfaces that transfer *raw data*. Such interfaces can be used for machine-to-machine communication or in the upcoming Web 2.0 applications based on asynchronous HTTP transfer using Javascript. The transferred data can be in a variety of formats (e.g., XML, JSON, or almost anything else) and we provide a mechanism to support multiple formats at once in a single web application. A client could even send a request in XML and receive the response in JSON format. The type-based combinators we presented to specify how data has to be converted are inspired by the WUI combinators presented by Hanus [5]. In fact they are so similar, that we think an integration of both approaches should be possible.

So, for future work we plan to integrate the machine-interface framework with the user-interface framework presented by Hanus [5]. We think, it should be possible to support the manipulation of local data through a web based interface by both humans and machines with a single web application.

Furthermore, the type-based combinators to construct XML converters serve well to model existing datatypes as XML data. However, modeling existing XML data as datatypes in Haskell or Curry is not always possible using our combinators. For example, attributes are not supported. We are working on a more flexible combinator library for XML converters and plan to integrate it into our framework for resource-based web applications.

7 CONCLUSIONS

We presented an approach to write resource-based web applications in a declarative programming language like Haskell or Curry.

We provide a datatype `Resource a b c` and library functions `get` and `update` to create a uniform interface to local and remote resources. Remote resources are directly accessed via HTTP and we provide operations to create a CGI program that makes local resources available over the Internet. Using our library both the client and server side of web applications can be implemented without knowledge of the details of HTTP communication. Hence, the programmer can concentrate on the application logic which is separated from network communication.

The user of our library can control the representation of resources during transfer and it is possible to support different formats in a single web application with content negotiation. We provide type-based combinators to concisely construct XML and JSON converters and integrate them seamlessly into our framework. Moreover, we implement authenticated communication over HTTP and integrate it transparently into our framework, i.e., hiding the details from the programmer of web applications.

We see two main advantages in using a declarative programming language like Haskell or Curry for writing web applications: Algebraic datatypes are tree-like structures and therefore well suited to represent hierarchically structured data, e.g., in XML format. Furthermore, the type systems of Haskell and Curry help to find errors in advance, which is very important in applications publicly available over the Internet. Due to the type-based conversion combinators and the built-in error handling mechanism, the programmer of a web application always gets type-correct inputs from clients. She is therefore released from the burden to write complex and error-prone code, that is not primarily considered with the application itself.

REFERENCES

[1] S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[2] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. June 1999.

[4] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[5] M. Hanus. Type-Oriented Construction of Web User Interfaces. Technical report, Technical Report, CAU Kiel, 2006.

[6] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[7] E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.

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

[9] P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In *4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*, pages 192–208. Springer LNCS 2257, 2002.

[10] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *International Conference on Functional Programming (ICFP'99)*, September 1999.