Chapter 1

Resource-Based Web Applications

Sebastian Fischer[1]

Abstract: We present an approach to write web applications in the functional logic language Curry. Logic features are employed to provide type based conversion combinators for arbitrary datatypes. With a restricted version of these combinators our library can also be implemented in a purely functional language.

The web applications we propose are directly based on the Hypertext Transfer Protocol (HTTP) – no additional protocol on top of HTTP is necessary. We provide a typed interface to HTTP that abstracts from the format used to transfer the data. Hence, we decouple a *resource* from its *representation* on the network.

1.1 INTRODUCTION

Present-day web applications are based on complex technologies (SOAP/WSDL) that are mastered by comprehensive frameworks in the mainstream programming worlds of .NET and Java. Inspired by [7], we show that it is possible to achieve similar goals with much simpler technology and less protocol overhead.

We develop a library for web applications based on simple HTTP message exchange without an additional protocol on top of HTTP. We provide functions to exchange arbitrary data as typed values, i.e., the user of our library is not concerned with error handling regarding data conversion. Most web applications use XML as platform independent format for data interchange and XML is tightly coupled with the web service frameworks mentioned above. With our library the representation of the transferred data is independent of the application itself. The programmer can plug in his own conversion functions into the framework. We even support multiple representations at once in a single web application. To simplify the definition of conversion functions, we provide type-based combinators to

[1]Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany;
E-mail: `sebf@informatik.uni-kiel.de`

construct converters for standard data formats. Upcoming so-called AJaX[2] applications often use Javascript Object Notation (JSON)[3] instead of XML. It is more convenient to program with native objects instead of a representation of XML data on the client side of such applications. We primarily support different representation formats to be able to develop the server side of such applications using our library.

The remainder of this paper is structured as follows: Section 1.2 introduces key concepts of the Internet architecture and sketches different architectural styles to design web applications. In Section 1.3 we present the API of our library and discuss an example how to use it in a client application. Section 1.4 focuses on data conversion and Section 1.5 introduces the part of our library that supports server applications. In Section 1.6 we give an overview of the implementation of our library. Related and future work is considered in Section 1.7 before we conclude in Section 1.8.

1.2 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) [8] – possibly employing some protocol on top of it. Note that we do not primarily consider applications running inside a web browser like [9, 12, 14].

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 [7] 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

[2]**A**synchronous **Ja**vascript and **X**ML

[3]`http://www.json.org/`

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 through a single URI. In the architectural style proposed by Fielding, a client sends an HTTP GET request to retrieve a representation of the resource. To update a resource, the client sends a POST request to the URI that corresponds to the resource. 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 functional (logic) programmers, hiding the details of HTTP communication. Hence, our framework is designed to support RESTful web applications. However it can also be used to model web applications of different paradigms (e.g. RPC-style) since it provides an interface to arbitrary HTTP communication via the GET or POST method.

1.3 A RESOURCE API

We assume that the reader is familiar with the syntax of the programming language Haskell [13] or Curry [11]. Like the Internet architecture sketched in the previous section, our library is built around the concept of *resources*. 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. The complete signature of `remoteResource` is given in Section 1.4.1.

Note that specifying a resource is not an IO action as no communication with the outside world is necessary. Only *access* to resources is done using IO actions.

1.3.1 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[4]

```
[NewsRef "Italy wins world cup final"
  "2006/7/9" "www.example.net/news/final"]
```

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

```
final :: Resource NewsItem Comment NewsItem
final = remoteResource "www.example.net/news/final"
```

and retrieve the item by `(get final)` which could return

```
NewsItem "Italy wins world cup final" "2006/7/9"
  "The winner is ..." []
```

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

```
update final "What a great match!"
```

and would get

```
NewsItem "Italy wins world cup final" "2006/7/9"
  "The winner is ..." ["What a great match!"]
```

as the result of this call, which is equal to the result of an immediately[5] following call to `(get final)`.

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

[5]if no other client changes the item in between

Consider the three arguments of the type constructor `Resource` given in the definition of `final`: 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 `NewsRef`s 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 "Next world cup in four years" "2006/7/10"
  "See you in four years ..." [])
```

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 "Next world cup in four years"
  "2006/7/10" "www.example.net/news/next-wc"
```

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 a function `authResource` that is similar to `remoteResource`. The function has two additional arguments of type `String` to specify the user name and password that identify the client.

1.4 DATA AND ITS REPRESENTATION

In Section 1.3 we omitted the details of data conversion for the transfer over the Internet. Unfortunately, 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. Curry provides 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 [5, 15]. 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.

JSON is another language independent data format 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.

1.4.1 Content-Type-based Conversion

The function `remoteResource` mentioned in Section 1.3 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)]
data ContentType  = Plain | Xml | Json |...| CT String
type ReadShow a   = (String -> Maybe a, a -> String)
```

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`[6] that converts values of arbitrary first order type[7]:

```
plainConv :: Conv a
plainConv = [(Plain, (safeRead, showQTerm))]

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

1.4.2 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` as part of our library. 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 `Bool` one label for each `True` and `False` is specified. More complex converters can be built 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)
```

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

[7]`readsQTerm` and `showQTerm` are standard Curry functions to read and show qualified data.

Since all first order datatypes can be mapped to values of types covered by the presented combinators, these are sufficient to build converters for such 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 fact, we can employ the concept of function patterns [4] 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. Note, that we employ the logic features of Curry to provide more convenient combinators. However, the expressiveness of the combinators is not changed, since similar converter specifications could be constructed using `cAdapt`.

A generalization of `cPair` is defined as part of our library using function patterns as[8]:

```
 c2Cons  :: (a -> b -> c) -> String
         -> ConvSpec a -> ConvSpec b
         -> ConvSpec c
 c2Cons 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]
```

Similar converters are defined for constructors that take one (`cCons`) or more (`c3Cons`, `c4Cons`, ...) arguments. As another logic feature of Curry we employ nondeterminism to specify converters for datatypes with multiple constructors. For example a converter specification for binary trees can be defined as:

[8] `xml tag xs = XElem tag [] xs` describes an XML element with empty attribute list.

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

cTree :: ConvSpec Tree
cTree = cCons Leaf "leaf" (cInt "value")
cTree = c2Cons 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 1.3.1:

```
cNewsRef :: ConvSpec NewsRef
cNewsRef = c3Cons NewsRef "ref"
 (cString "headline") (cString "date") (cString "uri")

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

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.

1.5 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 details of accessing a local resource are hidden inside the resource datatype. 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[9])) -> IO ()
```

These functions associate a resource with a URI to provide remote access. 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 1.6 after we reconsider the news example.

1.5.1 Example Continued: The News Provider

In Section 1.3.1 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:

```
 getNewsPaths   :: IO [Path]
 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 1.3.1. We define a function `newsProvider` that dispatches requests to appropriate handler functions:

```
 newsProvider :: Path -> IO (Maybe Handler)
 newsProvider p
  | p == "" = return (Just newsList)
  | otherwise = do
    ps <- getNewsPaths
    if elem p ps then return (Just (newsItem p))
     else return Nothing
```

[9]The type `Handler` is defined in Section 1.6.1.

```
newsList :: Handler
newsList = provide cNewsRefs cNewsItem cNewsRef
  (localResource getNewsList addItem)

newsItem :: Path -> Handler
newsItem path = provide cNewsItem cComment cNewsItem
 (localResource (getNewsItem path) (addComment path))
```

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.

1.6 IMPLEMENTATION

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

We represent HTTP messages as algebraic datatype `Http`. Since its definition is not important to understand the rest of this section, we do not discuss this datatype here.

1.6.1 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.

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[10] the location of the created CGI program points to a new resource. Unlike [12], we explicitly handle multiple URIs with a single CGI program. The most obvious type for a function that creates a CGI program from different resources associated with paths would be

```
cgi :: (Path -> Resource a b c) -> IO ()
```

This function would take a dispatcher mapping paths to resources and return an IO action that act as CGI program. Unfortunately, such a dispatcher cannot be defined. The value of type `Path` has no information about the type of the resource to be returned. Probably, existential quantification could be employed to use a list of paths associated to resources of arbitrary type instead of the dispatcher. Fortunately, we can define a similar `cgi` function without this feature: instead of returning a typed resource, the dispatcher supplied as argument to `cgi` 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. The dispatcher returns an optional handler to be able to indicate invalid path arguments. Also, it may perform IO actions to determine whether the path is valid or to compute an appropriate resource.

We now discuss how untyped handler functions can be created from typed resources. We need to define functions that handle GET and POST requests:

```
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.

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

- `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
```

So, in order to be able to define a dispatcher function as argument of `cgi`, we need to compute handlers that abstract from the types of the resource. A handler computes untyped HTTP messages using the converters that match the types of the associated resource. The higher-order features of the implementation language are essential for the described modeling of handler functions and dispatchers.

1.6.2 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 the functions

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

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

to create a handler that handles only GET requests and

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

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

to handle only POST requests. These functions generate a *Method Not Allowed* error message for other methods.

The type `Conv a` that specifies data conversion was defined in Section 1.4.1 as follows:

```
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.

In this section we sketched the implementation of a library for web applications implemented in the functional logic programming language Curry. Our library is directly based on HTTP and uses the logic features of Curry only for the type-based conversion combinators presented in Section 1.4.2. Since data conversion is explicit in our framework, resources can be transferred in arbitrary representations. The implementation employs higher-order features of Curry to provide a declarative interface for describing server applications.

1.7 RELATED AND FUTURE WORK

Hanus [9], Meijer [12] and Thiemann [14] provide frameworks to build server side web applications based on HTML forms that run inside a web browser. Although [12] abstracts from network communication, it uses strings as references to input fields, which is a source of possible errors. [9, 14] 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 [10] presented an approach to concisely describe type-based web user interfaces (WUIs), based on [9].

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 [10]. In fact they are so similar, that we plan to integrate both approaches in the future.

The type-based conversion combinators have a strong generic programming flavor although they do not rely on special features for generic programming. Conversion between XML and typed data using generic Haskell is explored in [5]. Our approach is different because it supports user defined datatypes instead of using a fixed one. However, generic programming features [3] can also be employed to map between user defined types and XML data [6] – as it is done for GUI components in [1]. The conversion in [5] is more general than our approach because it supports arbitrary XML data. For example, we do not support attributes. Such limitations will be subject of future work.

Since the logic features of Curry are only employed for data conversion which could be realized using generic programming in Clean, our library could be ported to Clean by replacing monadic IO with explicit passing of unique environments [2] – similar to the explicit passing of resources in our approach.

1.8 CONCLUSIONS

We presented an approach to write resource-based web applications in the declarative programming language 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.

Since our library is directly based on HTTP, it involves considerably less protocol overhead than SOAP-based web applications. Moreover, our approach to web applications differs from others in that it does not determine the format used to transfer data. We build upon the concept of *resources* and abstract from their *representations* in a type-safe way using explicit conversion functions. Our library even supports different formats in a single web application with content negotiation. For clients of a web application it is very convenient to be able to choose between different representations. While typical clients probably use XML data, Javascript-based web sites may prefer data in JSON format because it can be parsed directly into Javascript values. In a local setting, a platform dependent textual representation may be preferred for efficiency reasons. We provide type-based combinators to concisely construct XML and JSON converters and integrate them seamlessly into our framework. Hence, for standard formats the conversion functions need not be implemented by hand. 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 for writing web applications: algebraic datatypes are tree-like structures and therefore well suited to represent hierarchically structured data, e.g., in XML format. If the type-based conversion combinators are used, the strongly typed messages help to find errors in advance, which is very important in applications publicly available over the Internet. Even if custom parsers are employed, there remains the advantage that these can be developed and tested separately. 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] P. Achten, M. v. Eekelen, R. Plasmeijer, and A. v. Weelden. Programming Generic Graphical User Interfaces. Technical report, Nijmegen Institute for Computing and Information Sciences, Faculty of Science, University of Nijmegen, 2005.

[2] P. Achten, J. H. G. v. Groningen, and R. Plasmeijer. High Level Specification of I/O in Functional Languages. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 1–17, London, UK, 1993. Springer-Verlag.

[3] A. Alimarine and M. J. Plasmeijer. A Generic Programming Extension for Clean. In *IFL '02: Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, pages 168–185, London, UK, 2002. Springer-Verlag.

[4] 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.

[5] F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A type-preserving XML Schema-Haskell data binding. In Bharat Jayaraman, editor, *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages, PADL'04*, volume 3057 of *LNCS*, pages 71–85. Springer-Verlag, 2004.

[6] F. Atanassow and J. Jeuring. Inferring Type Isomorphisms Generically. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC'04*, volume 3125 of *LNCS*, pages 32–53. Springer-Verlag, 2004.

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

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

[9] 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.

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

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

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

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

[14] 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.

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