Lightweight Declarative Server-Side Web Programming

Michael Hanus

Institut für Informatik, CAU Kiel, 24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. Web interfaces are an important part of many applications but their implementation is full of pitfalls due to the client/server nature of web programming. This paper presents a lightweight approach to web programming based on a standard infrastructure, in particular, the common CGI protocol between client and server. No specific additions are necessary on the server side. Our approach exploits declarative programming features to provide a high-level API for server-side web scripting. This API allows to check many programming errors at compile time by using functional (static typing, higher-order functions) as well as logic (free variables) programming features. Together with further abstractions, like session handling, persistence, and typeful database access, it is used for non-trivial web applications.

1 Introduction

A web interface provides an easy access to software systems, since it does not require local software installations (one can assume that desktop computers as well as mobile devices are equipped with some web browser). Implementing a web interface for some application program can be challenging due to the client/server nature of web programming. The application program, running on a web server, has to generate an HTML page containing form elements. The client, using a web browser, fills the form with data and submits a request back to the server which creates and sends an answer to the client. Since the standard protocols, HTTP and the Common Gateway Interface (CGI), are stateless without a permanent connection between the server and the client, additional programming infrastructure is required for the application program.

There are various proposals to abstract from these raw protocols. Some approaches use specialized languages, like MAWL [16], DynDoc [20], or Links [4]. If the application is implemented in another language, the use of such languages causes a gap during software development. Therefore, many specific libraries have been developed for existing programming languages (e.g., [3,17,19,23,26]). Such libraries often support a convenient construction of web pages but provide only limited static checks for programming errors. For instance, web forms use string constants to identify input fields. If these string constants are used in the application program, typos in the strings are not detected at compile time and might lead to dynamic execution errors. Furthermore, a web form has two parts:

a program generating the form (or a static web page containing form elements) and a program executed when the form is submitted (specified by some URL in the form). Obviously, this is more complex and error-prone than implementing a graphical user interface (GUI) with event handlers for a desktop application.

To hide this complexity, one can try to support a continuation-based programming model for web forms, i.e., one can try to associate handlers to submit buttons (and similar interaction elements) which are responsible to process the event, e.g., to return a new web page with the computed result. These handlers can be implemented by processes running on the web server (servlets) and by encoding some required data as hidden fields in the web form [6,22]. This seems necessary due to the stateless nature of CGI. Since a CGI program running on a server is terminated after delivering a web page containing form elements, some resources must be kept on the server to answer form submissions. This programming model requires specific extensions on the web server (e.g., servlets) and/or permanent processes created by invoking a CGI script.[1]

In this paper we present a new approach to server-side web programming with a continuation-based programming model. It is lightweight from a programming point of view: the programmer implements the web form together with the event handlers in a single program rather than separating the application into different programs or scripts. The program runs on a standard web server without specific extensions. We show an implementation of our approach in the functional logic programming language Curry [13]. The combined features of Curry enable the implementation as a library and supports the checking of inconsistencies in web forms, like missing identifiers, at compile time. Thus, any application implemented in Curry (i.e., also functional or logic programming applications) can easily be equipped with a web interface by using our library.

Some characteristic features of our approach are:

– We use standard CGI without additional requirements on the web browser or web server extensions.
– The web server is not loaded with permanent processes for CGI interactions.
– Our model for web programming is continuation-based, i.e., a web form can contain any number of interaction elements with associated event handlers that are invoked when a client starts an activity.
– The API for our programming model is implemented as a Curry library without any language extension. The API supports compositionality (combine several forms in one web page) and ensures the consistency of web forms and their handlers at compile time.
– Our event handler model abstracts from the raw CGI protocol and interaction (which is implemented by environment variables and value decoding on the server side).

Note that our approach is oriented towards server-side web programming where the application data is stored on the server and accessed and manipulated via

[1] Actually, this was the approach taken in a very early continuation-based library for web programming in Curry [6]. The practical problems caused by the web-server processes motivated the current approach.

web browsers. Client-side web programming, where computations take place in the web browser, is an independent aspect not covered by our approach.

This paper is structured as follows. The next section provides a short overview of the main features of Curry as relevant for this paper. Sections 3 and 4 discuss our approach to model basic HTML documents and interactive web forms. Section 5 introduces a type model for web forms that allows to detect some inconsistencies at compile time. Section 6 shows the implementation of stateful web interactions via a session concept. The implementation of our library is sketched in Sect. 7. Useful extensions of our approach and related work are discussed in the the final sections before we conclude.

2 Functional Logic Programming with Curry

As mentioned above, the combined features of the declarative multi-paradigm language Curry [13] are important to provide the high-level approach to web programming described below. Since Curry is basically an extension of Haskell, we assume familiarity with functional programming and Haskell and review only those additional aspects of Curry which are necessary to understand our concept. More details about functional logic programming can be found in the surveys [2,9].

Curry amalgamates features from functional programming (demand-driven evaluation, strong typing, higher-order functions) and logic programming (computing with partial information, unification, constraints). The syntax of Curry is close to Haskell[2] [18]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [1]—a conservative extension of lazy functional programming and logic programming.

A Curry program consists of data type definitions (introducing *constructors* for the data types) and *functions* or *operations* on these types. As an example, we show the definition of two operations on lists: the well-known list concatenation and an operation `last` which exploits logic programming features to compute the last element of a list:

```
(++) :: [a] → [a] → [a]            last :: [a] → a
[]     ++ ys = ys                  last xs | _ ++ [e] == xs
(x:xs) ++ ys = x : (xs ++ ys)              = e  where e free
```

Note that, in contrast to Prolog, variables not occurring in the left-hand side of a rule must be declared as `free` (apart from anonymous variables, like "`_`"). Since "`++`" can be called with free variables in arguments, the condition in the rule of `last` is solved by instantiating `e` and the anonymous free variable "`_`" to appropriate values before reducing the call to "`++`". Free variables are also essential for our approach to web programming.

[2] Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("f e").

Curry has more features than described so far.[3] In this work, we exploit two of them. *Type classes* allow to express ad-hoc polymorphism in a structured manner [25]. We will use them to enforce structural constraints when using web forms. *Monadic I/O* [24] is a declarative concept to structure interactive programs by enforcing sequential evaluation by monadic operations. We will use it to access a possible state used in web forms. Since we assume familiarity with Haskell, we skip a detailed discussion of these concepts.

3 Modeling Basic HTML

Before describing our approach to implement dynamic web pages, we start by modeling basic HTML documents in Curry.

Since HTML documents have a tree-like structure, they can be represented in logic or functional languages in a straightforward manner [3,17]. Here, we define the type of basic HTML expressions in Curry as follows:

```
data BaseHtml = BaseText   String
              | BaseStruct String Attrs [BaseHtml]

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

Thus, a basic HTML expression is either a plain string (`BaseText`) or a structure consisting of a tag, a list of attributes (name/value pairs), and a list of HTML expressions contained in this structure.

Since writing HTML documents in a program could be tedious with this definition, one can define operations as useful abbreviations (`htmlQuote` transforms characters with a special meaning in HTML, like `<`, `>`, `&`, `"`, into their HTML quoted form):

```
htxt s = BaseText (htmlQuote s)       -- plain text
h1     = BaseStruct "h1"      []      -- level 1 header
strong = BaseStruct "strong"  []      -- important content
hrule  = BaseStruct "hr"      [] []   -- line break
...
```

A complete web page contains a title, optional parameters (e.g., cookies, style sheets), and a content (the actual library supports more alteratives, e.g., plain text documents):

```
data HtmlPage = HtmlPage String [PageParam] [BaseHtml]
```

As before, we add some useful abbreviations:

```
page title hexps = HtmlPage title [] hexps

headerPage title hexps = page title (h1 [htxt title] : hexps)
```

As an example, we define a web page with a simple multiplication table. The function

```
mult2html :: (Int,Int,Int) → [BaseHtml]
mult2html (x,y,z) = [htxt $ show x ++ " * " ++ show y ++ " = ",
                     strong [htxt $ show z], hrule]
```

[3] Actually, Curry is intended as an extension of Haskell although not all of the numerous features of Haskell are actually supported.

maps a triple of integers into a line showing their multiplication in HTML format. Then the operation

```
multPage :: HtmlPage
multPage = headerPage "Multiplication of Digits" $
  concatMap mult2html [ (x,y,x*y) | x <- [1..10], y <- [1..x] ]
```

exploits standard operations and list comprehensions to define our main web page. One can easily define a pretty-printing operation

```
showHtmlPage :: HtmlPage  →  String
```

which transforms an `HtmlPage` term into the corresponding HTML string. If we install a Curry program with the main operation

```
main = putStrLn $ showHtmlPage multPage
```

as a CGI executable on a web server, we get a simple dynamically generated web page. Since web server programs written with our library are always pages that are dynamically generated, we assume in the following that a dynamic web page is of type "`IO HtmlPage`".[4]

A CGI program generates a web page when a client accesses it. Hence, we can make it more dynamic by accessing data from its execution environment. For instance, the following page shows the current server time:

```
timePage :: IO HtmlPage
timePage = do time <- getLocalTime
              return $ headerPage "Current Server Time"
                         [htxt $ calendarTimeToString time]
```

4 HTML Forms

HTML forms are basic elements to interact with applications running on some server via a client's web browser. HTML forms are embedded in HTML pages and contain input elements to be filled out by the user and interaction elements (e.g., buttons) to submit the form data to a web server. When a form is submitted, the data contained in the input elements is encoded and sent to the server which starts a program, also called *form handler*, to react to the submission. The activated program decodes the input data, runs the application with the data, and returns an HTML page which is then sent back to the client.

The following HTML page contains a form with an input field to enter string:

```
<html>
  <head><title>String input form</title></head>
  <body>
    <form method="post" action="http://.../handler.cgi">
      Enter a string: <input type="text" name="FIELD1"/>
      <input type="submit" value="Submit"/>
    </form>
  </body>
</html>
```

[4] Actually, there is a simple script to wrap such web page definitions with `showHtmlPage` before compiling and installing it.

In a typical scenario, such a document is generated by some program. When the client presses the `Submit` button, the identifier `FIELD1` together with the contents of this text field is transmitted to the web browser which executes the program identified by the URL in the `action` attribute of the form.

In contrast to GUI programming for desktop applications, web interfaces are split into two independent parts accessible via different URLs: the first part generates the document containing the form, and the second part processes it and returns an answer document. Although there are many libraries and web frameworks to integrate these parts in one program from which these two parts are generated, there is the problem to ensure the consistency of both parts. Since input fields are part of the HTML structure and not program entities, their values are identified by strings, like `FIELD1`, so that the form handler has to access the actual input values via these strings. Thus, undefined input fields or typos in names of input fields are not detected at compile time but lead to execution errors or unintended behaviors.

The objective of our approach is to simplify the programming of web interfaces and make them more reliable by providing static consistency checks. As we will see, the functional and logic features of Curry are useful to support such a programming interface. In the following, we present our approach from a programmer's perspective before we discuss its implementation in Sect. 7.

Our programming model is based on two ideas:

1. Input elements are referenced by program variables rather than strings. Thus, typos in element names lead to accesses to undefined variables which will be reported at compile time.
2. Interaction elements, like submit buttons, are equipped with event handlers which process user inputs and return HTML documents. In order to access the actual form input, an event handler is invoked with an environment to look up the input data.

To refer to input elements, there is a type

```
data HtmlRef = HtmlRef String
```

This type is abstract, i.e., the constructor is not exported so that it cannot be used in an application program. Thus, the only way to use references in forms is via free variables. This is intended, as we will see later.

Input elements are constructed with references. For instance, a text input field is constructed by "`textField ref cont`" where `ref` is a reference of type `HtmlRef` and `cont` an initial contents. Since an interaction element, like a submit button, has an event handler, we have the following type synonyms:

```
type HtmlEnv = HtmlRef  →  String
type HtmlHandler = HtmlEnv  →  IO HtmlPage
```

An environment is just a mapping from references to strings and a handler maps an environment into an action which manipulates the server environment (e.g., database updates) and returns a new page. Then "`button s hdlr`" constructs a button with label `s` and associated handler `hdlr` (of type `HtmlHandler`). With these elements, we can define a form, as shown in Fig. 1, with a text input field

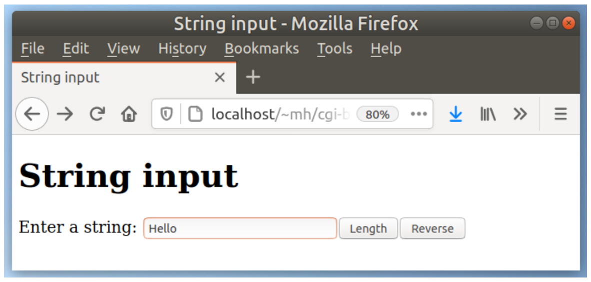

Fig. 1. An HTML page with a form to compute the length or reverse a string

and two submit buttons: one to compute the length of the string and another one to reverse the string.

```
lengthRevForm =
  [htxt "Enter a string: ", textField tref "",
   button "Length" lengthHandler, button "Reverse" revHandler]
 where
  tref free

  lengthHandler env = return $ page "Answer"
    [h1 [htxt $ "String length: " ++ show (length (env tref))]]

  revHandler env = return $ page "Answer"
    [h1 [htxt $ "Reversed input: " ++ reverse (env tref)]]
```

Since references to input elements are of type `HtmlRef` and there are no visible constructors of this type, only free variables can be used for this purpose. If these variables are in the scope of the event handler, one can easily access the actual input available when the handler is invoked by applying the environment to the reference (as with `(env tref)` above).

How can we invoke an event handler when a form is submitted? Note that the event handler is introduced with the form but the form submission takes place at some later point of time. One possibility is to start, together with the created form, a process which contains the code of the event handlers in this form. When a form is submitted, the input data is passed to this process. This has some similarity with servlets and was the basis of an early approach to HTML programming in Curry [6]. A disadvantage of that approach is the creation of many processes whose lifetime is not easy to determine, since form submissions might never take place. This could be improved by sharing processes, introducing timeouts etc, but it turned out to be a source of practical run-time problems in larger applications.

Another possibility is to submit the same program again but in a different mode: instead of producing the form elements, the event handler of the corresponding submit button is executed. In order to execute this handler, its code must be accessible from a top-level operation in the program. Therefore, we require that forms must be declared as top-level entities. In order to use forms in arbitrary HTML documents, we distinguish between a *form definition* and its actual use. To define a form as a top-level entity, there is a constructor `simpleFormDef` (later, we will see its concrete type and more complex form definitions) which wraps a form layout together with its event handlers into a form definition. For instance, we turn our form above into a form definition by

```
lengthRevFormDef = simpleFormDef lengthRevForm
```

We can *use a form* in an HTML document by wrapping the form definition with operation `formElem`. For instance, the following code defines an HTML page containing our form:

```
stringInputPage = return $
  headerPage "String input" [formElem lengthRevFormDef]
```

The advantage of distinguishing a form definition from its actual use is that we can use a form in any HTML document, in particular, recursively in the answer computed by an event handler. For instance, a form to compute the length of an input string and showing the form again in the answer can be defined as:

```
lengthForm = simpleFormDef
  [htxt "Enter a string: ", textField tref "",
   button "Length"
    (\env  →  return $ page "Answer"
                [h1 [htxt $ "Length: " ++ show (length (env tref))],
                 hrule, formElem lengthForm])]
 where tref free
```

Note that this compact definition is enabled by the combined logic and functional features of Curry: free variables, like `tref`, as "unknown" references instead of concrete strings, and event handlers, i.e., functions, in data structures. Next we show how to exploit advanced typing features from functional programming to make form programming more reliable.

5 Stronger Form Typing

Our modeling of forms assumes that input elements and submit buttons are used inside a form definition so that the corresponding event handlers can be invoked when the form is used. However, it seems that elements like `textField` or `button` can be mixed with basic HTML elements, like `htxt` or `strong`, so that they can also occur in basic HTML documents without an associated form. In order to avoid such problems, we exploit type classes to enforce more structure.

Basic HTML elements, like `htxt` or `h1`, can be used inside and outside forms, but input elements and submit buttons should be used only inside forms. This demands for some kind of overloading which is supported via type classes in a structured way [25]. Hence, we introduce a type class `HTML` which supports operations to construct HTML documents with textual and structured elements:[5]

```
class HTML a where
  htmlText   :: String  →  a
  htmlStruct :: String  →  Attrs  →  [a]  →  a
```

[5] The actual definition in our library contains more operations which are not relevant here.

The type of basic HTML expressions is an instance of this class:

```
instance HTML BaseHtml where
  htmlText   = BaseText
  htmlStruct = BaseStruct
```

To model form elements, we introduce an extended data type for general HTML expressions which also contains alternatives for elements with references (input elements) and elements with event handlers (e.g., buttons).

```
data HtmlExp = HtmlText   String
             | HtmlStruct String Attrs [HtmlExp]
             | HtmlInput  HtmlRef HtmlExp
             | HtmlEvent  HtmlRef HtmlHandler HtmlExp
```

The actual use of the additional constructors to model input and interaction elements is not relevant here. For the moment it is only important that input and interaction elements are of type `HtmlExp` rather than `BaseHtml`, e.g., the types of text input fields and buttons are

```
textField :: HtmlRef  →  String  →  HtmlExp
button    :: String   →  HtmlHandler  →  HtmlExp
```

Now we can show the actual types of the operations to define simple forms and to use them (the type constructor `HtmlFormDef` will be discussed later):

```
simpleFormDef :: [HtmlExp]  →  HtmlFormDef ()
formElem      :: HtmlFormDef a  →  BaseHtml
```

Thanks to these type signatures, input and interaction elements (of type `HtmlExp`) can be used inside a form definition but not in basic HTML documents outside form elements. Moreover, forms cannot be nested, as required by the HTML standard,[6] since the result type of `formElem` is `BaseHtml` rather than `HtmlExp`.

In order to use other basic HTML elements inside and outside forms, we define the type of general HTML expressions also as an instance of class `HTML`:

```
instance HTML HtmlExp where
  htmlText   = HtmlText
  htmlStruct = HtmlStruct
```

Finally, we redefine the abbreviations for basic HTML elements introduced in Sect. 3 by overloading them with type class `HTML`:

```
htxt :: HTML h => String  →  h
htxt s = htmlText (htmlQuote s)

h1 :: HTML h => [h]  →  h
h1 = htmlStruct "h1" []
...
```

Hence, we can use these elements inside and outside HTML forms, but the definition of an HTML page with buttons outside forms, like

```
page "Illegal" [..., button "Submit" ...]
```

leads to a static type error.

[6] https://html.spec.whatwg.org/multipage/forms.html#the-form-element

6 Stateful Forms

The HTML forms presented so far are quite limited. Form definitions are top-level entities. Therefore, data from the context of the form (e.g., database entities, authentication data) cannot be used in the form. Hence, we need to extend forms with the possibility to access some state. To show such *stateful forms*, it is only necessary to read data, whereas the manipulation of data is usually performed by the event handlers (therefore, event handlers are of type `IO`). For this purpose, there is a monad `FormReader` with operations to read data (see below), i.e., the `FormReader` monad is a restriction of the `IO` monad where only read operations are supported.

To define a stateful form, one has to provide a `FormReader` action to read data of some type `a` and an operation which maps values of this type into an HTML form. Hence, a stateful form can be defined by the operation

```
formDef :: FormReader a → (a → [HtmlExp]) → HtmlFormDef a
```

Since the data might be read several times (to construct the form layout, to start an event handler, or if the form has multiple occurrences in a web page), it is important to use the restricted `FormReader` monad for data access instead of general `IO` actions. Therefore, an operation of type `HtmlFormDef a` defines a form which reads some data of type `a` and use this data to generate the actual HTML form.

Although HTTP is a stateless protocol, typical web applications require a session concept to pass information between different web pages, like the login name of a user or the contents of a virtual shopping basket. A session concept can be implemented via cookies to identify a client in a session. The session information itself should be stored in the server for security reasons [14]. In order to hide the details of session handling, there is a library (more details can be found in [11]) which provides the following operations (the type variable `a` denotes the type of session data):

```
getSessionData    :: Global (SessionStore a)  → a → FormReader a
putSessionData    :: Global (SessionStore a)  → a → IO ()
removeSessionData :: Global (SessionStore a)  → IO ()
```

Here, "`Global (SessionStore a)`" is the type of a top-level entity referring to some cell with session information of type `a`. `getSessionData` retrieves information of the current session (and returns the second argument if there is no information, e.g., in case of a new session), `putSessionData` stores information in the current session, and `removeSessionData` removes such information.

In order to see an application of this concept, we implement a number guessing game: the client has to guess a number known by the server, and the server responds whether the client's number is smaller, larger, or correct. In the latter case, the number of trials is shown.[7] Hence, the session state contains the number of trials and has the following type:

```
trials :: Global (SessionStore Int)
```

[7] Of course, this game can be implemented on the client side, but a realistic example with database access needs too much space.

The form definition consists of an action that reads the current session data and the HTML form for this data:

```
guessForm = formDef (getSessionData trials 1) guessFormHtml

guessFormHtml t = [htxt "Guess a number: ", textField nref "",
                   button "Check" guessHandler]
 where
  nref free

  guessHandler env = do
    let g = read (env nref)
    if g == 42
      then do removeSessionData trials
              return $ headerPage
                ("Correct! " ++ show t ++ " guesses!") []
      else do putSessionData trials (t+1)
              return $ headerPage
                ("Too " ++ if g<42 then "small" else "large")
                [formElem guessForm]
```

The form handler reads the user input from the environment (one could add an additional check whether the input is a number string) and compares it with the "secret" number. If it is equal, the session data is removed before returning the answer, otherwise the session data is updated so that the next form invocation gets the updated data.

7 Implementation

The main objective of this work is to provide an approach to server-side web programming that is easy to use. We have already seen that event handlers for interaction elements, free variables to identify input elements, and advanced typing leads to a compact and reliable programming model. Now we discuss the implementation of our approach. In order to keep it also lightweight and easy to use, we base it on existing interfaces provided by any common web server. For this purpose, we show how to compile a Curry program containing the definition of a web page and various forms into a single executable to be installed as a CGI program on a web server.

As an example, consider the compilation of `stringInputPage` of Sect. 4. When the generated CGI program is invoked, it has to write the HTML text of the page described by this operation on the standard output. In order to avoid installing various CGI programs or forking processes on the server, the same executable is invoked when the form is submitted by one of the two submit buttons. Thus, the executable has to check the context of its invocation in order to choose the right behavior. For this purpose, form elements are translated at run time as follows:

1. Each form contains a hidden field `FORMID` with the unique name of the form (the qualified name of the defining operation).
2. All input and interaction elements have unique identifiers. These identifiers are generated at run time (when the form is computed) by instantiating free `HtmlRef` variables with unique strings.

3. When the CGI program is invoked, it checks whether the field `FORMID` is set. If not, the main page is generated, otherwise the corresponding form is executed.

For instance, the HTML element "`formElem lengthRevFormDef`" is translated into

```
<form method="post" action="?">
  <input type="hidden" name="FORMID" value="LR.lengthRevFormDef"/>
  Enter a string: <input type="text" name="FIELD_0" value=""/>
  <input type="submit" name="FIELD_1" value="Length"/>
  <input type="submit" name="FIELD_2" value="Reverse"/>
</form>
```

The actual implementation combines a standard Curry compiler with a simple preprocessor[8] (`curry2cgi`) that wraps the operation defining the main page and all form definitions with a dispatcher of type

```
printMainPage :: [(String, [(String,String)]  → IO ())]
              → IO HtmlPage  → IO ()
```

The first argument is a list of pairs consisting of a form identifier and a form implementation. The latter is an operation which takes a list of name/value pairs (the actual form inputs passed by CGI). For instance, if the operation `stringInputPage` is defined in program `LR`, this operation is compiled into

```
main = printMainPage
         [("LR.lengthRevFormDef", execFormDef lengthRevFormDef)]
         stringInputPage
```

which is finally compiled by the Curry compiler into a CGI executable.

The operation `execFormDef` translates a form definition into an operation which takes the CGI name/value pairs and write the HTML text produced by the corresponding event handler on the standard output. For this purpose, `execFormDef` executes the `FormReader` action of the form definition to read the required data, constructs the HTML form of the form definition to find the corresponding event handler and executes this handler by transforming the CGI name/value pairs into a `HtmlEnv` mapping.

The actual implementation of our library and the `curry2cgi` preprocessor is available as a Curry package (`html2`) which also contains the examples shown in this paper. The library and the preprocessor exploit only standard features of Curry. The preprocessor uses libraries for meta-programming to read and represent source programs as abstract syntax trees in order to collect the forms defined in a program, and also performs some checks to ensure the correct use of the library at compile time, e.g., whether the operation `formElem` is applied to top-level entities.

8 Extensions

The programming interface for HTML forms is a basis on which further abstractions can be added. Some abstractions have been proposed for an earlier (less

[8] The preprocessor is used since contemporary Curry implementations do not provide access to entities of the program at run time.

reliable) approach to HTML programming [6]. Since they have been adapted to this new approach, we sketch them in the following.

As apparent from the definition of `HtmlEnv`, the input elements have always string values so that other kinds of values, like numbers, emails, URLs, or structured data, must be extracted from strings. Since this requires some (tedious) code for checking the validity of values in input fields, providing appropriate error messages, etc, a more abstract layer to construct *web user interfaces* (*WUI*s) in a type-oriented manner is proposed in [7]. Using WUIs, one can construct for each type of an application program a WUI which implements a web-based interface to manipulate values of this type. For instance, the corresponding library contains predefined WUIs to manipulate strings (`wString`) or to select a value (`wSelect`) from a given list of values (where the first argument shows a value as a string), where "`WuiSpec a`" denotes the type of a WUI to modify values of type `a`:

```
wString :: WuiSpec String
wSelect :: (a → String) → [a] → WuiSpec a
```

To construct WUIs for complex data types, there are *WUI combinators* that map simpler WUIs to WUIs for structured types. For instance, there is a family of WUI combinators for tuple types:

```
wPair   :: WuiSpec a → WuiSpec b → WuiSpec (a,b)
wTriple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)
...
```

Thus, "`wPair wString (wSelect show [1..31])`" defines a WUI to manipulate a pair of a string and a number between 1 and 31. WUIs can easily be adapted to specific requirements. For instance, one can attach a predicate so that the resulting WUI accepts only values satisfying this predicate. Thus,

```
wString `withCondition` (not . null)
```

specifies a WUI that accepts only non-empty strings. There are further combinators to change the default rendering or error messages and to transform a WUI into an HTML form to be embedded in web pages.

Based on WUIs and abstractions for typeful database programming [12], there is a web framework to generate a complete web-based system to manipulate data stored in relational databases from an entity-relationship (ER) model of the data [11]. The generated system supports authentication, authorization, session handling, and ensures the consistency of the database w.r.t. the data dependencies specified in the ER model. Since the framework generates high-level Curry code, it can easily be adapted to individual customer requirements.

To support domain-specific syntax in Curry programs, there is a preprocessor for Curry programs which replaces domain-specific syntax by standard Curry syntax. For instance, one can embed HTML fragments in Curry programs (instead of expressions constructed by `htxt`, `h1`, `strong`, etc) and one can also write database queries in SQL syntax which is checked for type consistency against a given ER model [12].

Our model for web programming and the extensions described above have been used for non-trivial web applications, like the curricula and module information system of our department[9] or Smap,[10] a web-based system to write, store, and execute programs in various programming languages.

9 Related Work

Due to the importance of web programming, there exist for almost any programming language various approaches to implement dynamic web pages. In the following, we discuss approaches related to declarative languages.

There are many approaches to hide low-level details of HTML and CGI programming in purely functional languages. For instance, Meijer [17] presents a Haskell library to free the programmer fom parsing and printing CGI-based interactions. Thiemann [22,21] proposes typed representations of HTML (and XML) documents by exploiting the type class system of Haskell. This can be considered as a further refinement of our representation of basic HTML documents and could also be added to our library. WASH [23] adds a session concept as a further abstraction in order to support server-side web applications. It is based on CGI and uses hidden input fields in HTML documents in order to add session-based stateful computations to CGI, similarly (but technically a bit different) from the early HTML library of Curry [6]. Since storing information in hidden input fields could be a source to attack web systems [14], we avoid them in our approach by taking the price into account that all data must be stored as session data kept on the server side.

The iData toolkit [19] supports the construction of web forms in a type-safe and declarative programming style based on purely functional programming. This toolkit follows the model-view-controller pattern and provides specific abstractions to manipulate data in web forms, similarly to the type-oriented web user interfaces library for Curry [7].

The Haskell library Haste.App [5] supports an approach to write web applications as a single program where type annotations are used to determine which parts are executed on the client (web browser) or on the server. Thus, separate server executables and JavaScript code is generated from a single program using this library and two compilers. This library exploits Haskell's type system to implement server and client code in a single program and ensures a type-safe communication between the client and server side. Since this approach concentrates on an elegant model for the communication aspects of web applications, the integration into web documents is done in the traditional way, i.e., by using raw strings to identify input elements.

There are also proposals for specific programming languages for web applications. For instance, Links [4] supports the implementation of a web application in a single program from which server and client side code (O'Caml and JavaScript programs) as well as database code (SQL) is generated. Being a specialized language, Links has some support to attach program variables to input fields so that their correct use is checked by the Links compiler. However, this input field

[9] `https://mdb.ps.informatik.uni-kiel.de/`

[10] `https://smap.informatik.uni-kiel.de/`

checking is done by the compiler so that general form abstraction and composition is not supported.

The PLT Scheme Web Server [15] is an approach to write web applications in PLT Scheme. Since the complete web server is implemented in PLT Scheme and supports servlets running on the web server, one can implement web pages and form interactions in a single program. Due to the dynamic nature of PLT Scheme, the consistency of input fields and their use in form handlers is not checked at compile time.

Related to logic programming, there are less advanced approaches. There are libraries for converting HTML documents to data terms so that convenient Prolog notation can be used to describe HTML documents [26]. The PiLLoW library [3] also supports dynamic web pages by providing abstractions for HTML forms and CGI communication. Due to the dynamically typed nature of Prolog, static checks on the form of HTML documents are not supported so that this library provides rather basic abstractions for web programming.

10 Conclusions

We presented a new approach to server-side web programming in the declarative language Curry. The approach is lightweight since its implementation is based on a library together with a wrapper for the main operation, and the execution of applications using this library does not require specific extensions for a web server rather than the ability to execute CGI programs. Our interface for web programming exploits the combined functional and logic features provided by Curry: free variables as references to input elements, functions in data structures as event handlers for interaction elements, and strong typing to check the consistent use of these elements at compile time. As a result, we obtain a more reliable and declarative approach to web programming compared to other alternatives.

Our approach is focused on the server side, i.e., it is intended to implement web interfaces to an application keeping their data on a server. Our model returns for each user request a new web page containing the answer information. Nevertheless, it could also be combined with client-side programming by generating JavaScript. Conceptually, this was tried for Curry with web user interfaces that perform immediate checks by compiling parts of the Curry program to JavaScript [8] or by a generic concept for declarative user interfaces that can be compiled as a desktop or as a web application [10]. For future work, we want to extend our implementation so that event handlers do not return complete HTML pages but only updates to be performed on the current page via DOM and Ajax.

Another line for future work is the integration of our programming model in a web browser so that the program is not started for every interaction in order to reduce the startup time. Nevertheless, our current implementation is sufficient for systems with a limited load, as demonstrated by existing applications mentioned in this paper.

References

1. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
2. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
3. D. Cabeza and M. Hermenegildo. Distributed WWW programming using (CIAO-)Prolog and the PiLLoW library. *Theory and Practice of Logic Programming*, 1(3):251–282, 2001.
4. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*, pages 266–296. Springer LNCS 4709, 2006.
5. A. Ekblad and K. Claessen. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, pages 79–89. ACM Press, 2014.
6. 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.
7. M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.
8. M. Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 155–166. ACM Press, 2007.
9. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
10. M. Hanus and C. Kluß. Declarative programming of user interfaces. In *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL'09)*, pages 16–30. Springer LNCS 5418, 2009.
11. M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. *Theory and Practice of Logic Programming*, 14(3):269–291, 2014.
12. M. Hanus and J. Krone. A typeful integration of SQL into Curry. In *Proceedings of the 24th International Workshop on Functional and (Constraint) Logic Programming*, volume 234 of *Electronic Proceedings in Theoretical Computer Science*, pages 104–119. Open Publishing Association, 2017.
13. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
14. S.H. Huseby. *Innocent Code: A Security Wake-Up Call for Web Programmers*. Wiley, 2003.
15. S. Krishnamurthi, J.A. McCarthy, P.T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT scheme web server. *Higher Order and Symbolic Computation*, 20(4):431–460, 2007.
16. D.A. Ladd and J.C. Ramming. Programming the web: An application-oriented language for hypermedia service programming. *World Wide Web Journal*, 1(1), 1996.
17. E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.

18. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
19. R. Plasmeijer and P. Achten. iData for the world wide web - programming interconnected web forms. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 242–258. Springer LNCS 3945, 2006.
20. A. Sandholm and M.I. Schwartzbach. A type system for dynamic web documents. In *Proc. of the 27th ACM Symposium on Principles of Programming Languages*, pages 290–301, 2000.
21. P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4-5):435–468, 2002.
22. 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.
23. P. Thiemann. WASH server pages. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 277–293. Springer LNCS 3945, 2006.
24. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.
25. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL'89*, pages 60–76, 1989.
26. J. Wielemaker, Z. Huang, and L. van der Meij. SWI-Prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.