Putting Declarative Programming into the Web: Translating Curry to JavaScript *

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

Abstract

We propose a framework to construct web-oriented user interfaces in a high-level way by exploiting declarative programming techniques. Such user interfaces are intended to manipulate complex data in a type-safe way, i.e., it is ensured that only type-correct data is accepted by the interface, where types can be specified by standard types of a programming language as well as any computable predicate on the data. The interfaces are web-based, i.e., the data can be manipulated with standard web browsers without any specific requirements on the client side. However, if the client's browser has JavaScript enabled, one could also check the correctness of the data on the client side providing immediate feedback to the user. In order to release the application programmer from the tedious details to interact with JavaScript, we propose an approach where the programmer must only provide a declarative description of the requirements of the user interface from which the necessary JavaScript programs and HTML forms are automatically generated. This approach leads to a very concise and maintainable implementation of web-based user interfaces. We demonstrate an implementation of this concept in the declarative multi-paradigm language Curry where the integrated functional and logic features are exploited to enable the high level of abstraction proposed in this paper.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.2 [*Software Engineering*]: Design Tools and Techniques—User interfaces; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; D.3.4 [*Programming Techniques*]: Processors—Compilers; H.5.2 [*Information Interfaces and Presentation*]: User Interfaces

General Terms Languages

Keywords Functional Logic Programming, User Interfaces, Web Programming, Curry, JavaScript

* This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-2.

PPDP'07 July 14–16, 2007, Wroclaw, Poland.

1. Motivation

The implementation of software systems can be coarsely classified into two parts: the implementation of the application logic and the implementation of the user interface. If one uses declarative programming languages, the first part can be implemented with reasonable efforts (in particular, if one uses libraries with appropriate interfaces, see, for instance, [10, 12, 18] for database programming). In contrast, the construction of the user interface is usually complex and tedious. In order to simplify the latter task, scripting languages with toolkits and libraries, like Tcl/Tk, Perl, or PHP, are one approach to support this goal. Since scripting languages often lack support for the development of complex and reliable software systems (e.g., no static type and interface checking, limited code reuse due to the lack of high-level abstractions), they are often used to implement the user interface whereas the application logic is implemented in some other language. This approach creates new problems since it is well known that such combinations could cause security leaks in web applications [25]. Therefore, an alternative solution is the embedding of such domain-specific languages in high-level languages able to provide appropriate abstractions (e.g., see [9, 17, 29, 32, 34, 35] for functional and logic languages). This paper follows the latter alternative and considers an approach to construct web user interfaces (WUIs) in a declarative way. The core idea of the declarative construction of WUIs has been presented in [21]. In this paper we combine this construction with the existing features of scripting languages by compiling parts of the declarative interface specifications into a scripting language available in almost all web browsers: JavaScript.

In order to provide an example of the new approach presented in this paper, we give a short summary of the framework to construct WUIs presented in [21]. This approach to construct WUIs is useful in situations where data of an application program should be manipulated via standard web browsers (i.e., by HTML forms). The application program supplies the WUI with the current data of the application and an operation to store the modified data. Furthermore (and most important), it provides a type-oriented specification of the WUI structure that matches the type of the application data. For this purpose, the WUI framework contains a set of *basic WUIs* to manipulate data of basic types, e.g., integers, truth values, strings, finite sets, and a set of *WUI combinators* to construct WUIs for complex data types from simpler types similarly to type constructors in programming languages. For instance, there are combinators for tuples, lists, union types etc. The framework ensures that the user inputs only *type-correct data*, i.e., if the user tries to input illegal data (e.g., incorrect integer constants, empty strings, wrong dates or email addresses), the WUI does not accept the data and asks the user to correct the input. Figure 1 shows an example of a WUI for a list of persons containing various input errors. Note that errors can occur not only in individual input fields but also as illegal combinations of different fields, like the date in

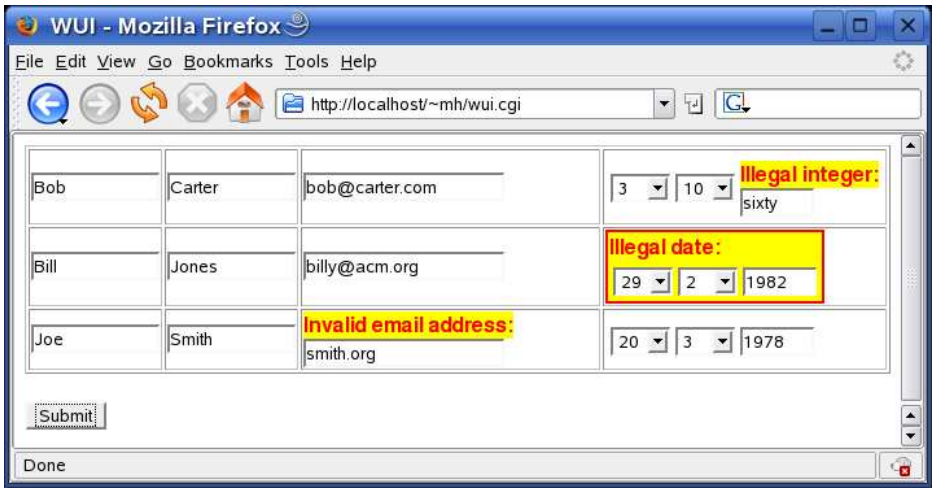

Figure 1. A WUI for a list of persons containing various input errors

the second row. Thanks to this feature of the WUI framework, the application program need not check the input data and perform appropriate actions (e.g., providing error forms to correct the input etc). This considerably simplifies the task of programming the user interface.

We have already mentioned that type-correct inputs have to be understood in a much broader sense than types used in programming languages. For instance, strings containing email addresses must have a particular form, a date like "February 29, 2006" is illegal, and two input fields containing a password and the repeated password must be always identical. Therefore, WUIs can be specified together with *any computable predicate* so that input data is only accepted if it satisfies the specified predicate. Furthermore, WUIs can be customized in various ways, e.g., to provide *application-specific error messages* in case of illegal inputs, to use a specific rendering or embed them in web pages with a specific layout, and can be easily adapted to user-defined types (see [21] for details). In order to provide a compact and high-level interface for the WUI construction, a concrete implementation has been performed with the declarative multi-paradigm language Curry [16, 24]. Various features of Curry, like the treatment of functions as first-order objects, logic variables, and strong typing are exploited in this implementation, although not all of them are required to be used by the programmer constructing WUIs. In particular, logic variables are used as internal references to input fields which are not visible to the programmer.

In this paper we show how to exploit the existence of JavaScript interpreters in web browsers in order to increase the functionality of WUIs. By translating conditions in WUIs into JavaScript programs, one can check user inputs on the client side, i.e., forms with illegal inputs are not sent to the web server. This feature reduces the number of client/server interactions and provides instantaneous feedback on incorrect inputs on the client side. However, Curry is a powerful language with advanced programming features (e.g., higher-order functions, laziness, logic variables, constraint solving, concurrency). Thus, it is not reasonable to translate into JavaScript all possible conditions that can be implemented in Curry, since this might finally require to communicate a complete Curry implementation to the web client. This is not only inefficient (since JavaScript is usually interpreted) or impossible (due to space and time limitations of the JavaScript interpreter) but also not necessary: the correctness of the user input is always checked on the server side (due to the principle in web programming that one should *never trust user inputs from web browsers* even if they are checked by scripts on the client side, since one never knows whether the input comes from a human using a web browser or another malicious program [25]). Thus, one is free to select only particular conditions that are easy to translate into JavaScript. This is the idea used in the following in order to get a reasonable and practically applicable combination of two different worlds of programming.

In the next section, we review the concepts of Curry, HTML programming, and JavaScript that are necessary to understand the contents of this paper. Section 3 reviews the ideas of WUIs. The basic ideas behind the combination of WUIs and JavaScript are described in Section 4 before the translation of Curry into JavaScript is defined in Section 5. Section 6 introduces some important optimizations. Section 7 describes the current implementation. Section 8 discusses related work before we conclude in Section 9.

2. Basics: Curry, HTML, JavaScript

2.1 Functional Logic Programming

Functional logic languages [15] integrate important features of functional and logic languages in order to provide a variety of programming concepts. Modern languages of this kind [16, 24, 28] combine the concepts of demand-driven evaluation and higher-order functions from functional programming with logic programming features like computing with partial information (logic variables), unification, and nondeterministic search for solutions. This combination, supported by optimal evaluation strategies [4] and new design patterns [5], leads to better abstractions in application programs, e.g., as shown for programming with databases [12, 18] or dynamic web pages [17, 21]. The declarative multi-paradigm Curry [16, 24] is a functional logic language extended by concurrent programming concepts and has been used in various applications. In the following, we review the elements of Curry relevant for this paper but omit other features not used here (e.g., constraints, search strategies, concurrency, I/O concept, modules). More details about Curry's computation model and a complete description of all language features can be found in [16, 24].

From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [30], i.e., (type) 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 ("$f\ e$"). A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are first-order citizens as in Haskell and are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. Function calls with free variables are evaluated by a possibly nondeterministic instantiation of demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule.

In general, functions are defined by (*conditional*) *rules* of the form "$f\ t_1 \ldots t_n\ \texttt{|}\ c\ \texttt{=}\ e$" with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the (optional) *condition* c is a constraint (e.g., a conjunction of equations), and e is a well-formed *expression* which may also contain function calls, lambda abstractions etc. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable.

EXAMPLE 2.1. *The following Curry program defines the data types of Boolean values and polymorphic lists, and functions to compute the concatenation of lists and the last element of a list:*

```
infixr 5 ++

data Bool   = True    | False
data List a = []      | a : List a

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] -> a
last xs | ys ++ [x] =:= xs   = x   where x,ys free
```

`[]` *(empty list) and* `:` *(non-empty list) are the constructors for polymorphic lists (*`a` *is a type variable ranging over all types and the type "*`List a`*" is written as* `[a]` *for conformity with Haskell). The* infix operator declaration "`infixr 5 ++`" *declares the symbol "*`++`*" as a right-associative infix operator with precedence* `5` *so that we can write function applications of this symbol with the convenient infix notation. The (optional) type declaration ("*`::`*") of the function "*`++`*" specifies that "*`++`*" takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.*[1] *Since the function "*`++`*" can be called with free variables in arguments, the equation "*`ys ++ [x] =:= xs`*" is solved by instantiating the first argument* `ys` *to the list* `xs` *without the last argument, i.e., the only solution to this equation satisfies that* `x` *is the last element of* `xs`*. In order to support some consistency checks,* extra variables*, i.e., variables of a rule not occurring in a pattern of the left-hand side, must be declared by "*`where...free`*" (see the rule defining* `last`*).*

The operational semantics of Curry, described in detail in [16, 24], is based on an optimal evaluation strategy [4] which is a conservative extension of lazy functional programming and (concurrent) logic programming. Due to its demand-driven behavior, it provides optimal evaluation (e.g., shortest derivation sequences, minimal solution sets) on well-defined classes of programs (see [4] for details). Curry also offers standard features of functional languages, like higher-order functions or monadic I/O (which is identical to Haskell's I/O concept [36]).

[1] Curry uses curried function types where $\alpha\texttt{->}\beta$ denotes the type of all functions mapping elements of type α into elements of type β.

2.2 HTML Programming

Writing programs that generate HTML documents requires a decision about the representation of HTML documents. A textual representation (as often used in CGI scripts written in Perl or with the Unix shell) is very poor since it does not avoid certain syntactical errors (e.g., unbalanced parenthesis) in the generated document. Thus, it is better to introduce an abstraction layer and model HTML documents as elements of a specific data type together with a wrapper function that is responsible for the correct textual representation of this data type. Since HTML documents have a tree-like structure, they can be represented in functional or logic languages in a straightforward way [9, 17, 29, 34]. For instance, the type of HTML expressions is defined in Curry as follows:[2]

```
data HtmlExp =
     HtmlText   String
   | HtmlStruct String [(String,String)] [HtmlExp]
```

Thus, an HTML expression is either a plain string or a structure consisting of a tag (e.g., `b`,`em`,`h1`,`h2`,...), a list of attributes (name/value pairs), and a list of HTML expressions contained in this structure. Thus, the HTML code

```
<p>This is an <i>example</i></p>
```

is represented by the data term

```
HtmlStruct "p" []
 [HtmlText "This is an ",
  HtmlStruct "i" [] [HtmlText "example"]]
```

Since it is tedious to write HTML documents in this form, one can provide various functions as useful abbreviations (`htmlQuote` transforms characters with a special meaning in HTML into their HTML quoted form):

```
htxt   s     = HtmlText   (htmlQuote s)
par    hexps = HtmlStruct "p" [] hexps
italic hexps = HtmlStruct "i" [] hexps
...
```

Then we can write the example as

```
par [htxt "This is an ", italic [htxt "example"]]
```

A *dynamic web page* is an HTML document (with header information) that is computed by a program at the time when the page is requested by a client (usually, a web browser). For this purpose, there is a data type

```
data HtmlForm =
     HtmlForm String [FormParam] [HtmlExp]
```

to represent complete HTML documents, where the first argument to `HtmlForm` is the document's title, the second argument contains optional parameters (e.g., cookies, style sheets), and the third argument is the document's content. Since a dynamic web page should represent information that often depends on the environment of the web server (e.g., stored in databases), a dynamic web page has always the type "`IO HtmlForm`", i.e., it is an I/O action [36] that retrieves some information from the environment and produces a web document.

Dynamic web pages usually process user inputs, placed in various input elements (e.g., text fields, text areas, check boxes) of an HTML form, in order to generate a user-specific result. For this

[2] This definition covers only the tree-like structure of HTML documents but does not enforce further restrictions. Hence, documents not conforming with the HTML standard can be created. This can be avoided with refined definitions and more sophisticated type structures [34].

purpose, the HTML library of Curry [17] provides an abstract programming model that can be characterized as *programming with call-back functions*. A web page with user input and buttons for submitting the input to a web server is modeled by attaching an *event handler* to each submit button that is responsible for computing the answer document. In order to access the user input, the event handler has access to an environment containing the actual user input. We omit further details here (they can be found in [17]) since we consider a more abstract layer to construct web-based user interfaces that will be described in Section 3. We only want to remark that the functional as well as logic features of Curry are exploited to implement this programming model: event handlers and environments are functions attached to data structures representing HTML documents, and input elements in a document have logic variables as references to support consistency checks by the compiler (in contrast to the use of strings in traditional web scripting, e.g., raw CGI, Perl, PHP).

2.3 JavaScript

JavaScript [13] is an imperative scripting language that can be embedded in HTML documents. JavaScript programs are executed by the client's web browser and have access, via a document object model, to the resources of the browser, in particular, to the HTML document shown in the browser. For this purpose, the document is represented as a hierarchical object structure where the attributes of each object can be accessed or manipulated by the standard "dot notation". For instance, the class identifier (whose meaning is usually defined in a style sheet) of an object `elem` in an HTML document can be changed to `myStyle` by the assignment `elem.className = "myStyle"`.

JavaScript programs are usually executed by the web browser when some event occurs. For instance, if a text input field in an HTML form has an attribute `onblur="f(3)"`, the function call `f(3)` is evaluated whenever the user leaves this input field. In this paper, we exploit this functionality of JavaScript to check the user input on the client side before the complete web form is submitted to the server.

3. Type-Oriented Web User Interfaces

In this section we review the type-oriented construction of WUIs, as proposed in [21], from a programmer's point of view, before we discuss its extension to include JavaScript in the next section.

The basic idea of our WUI framework is to provide an easily applicable method to implement an interface for the manipulation of data of an application domain. Thus, we assume that the application program supplies a WUI with the current state of the data and an operation to store the data modified by the user and acknowledge it to the user. Thus, the main operation to construct a WUI has the type signature (remember that dynamic web pages are of type `IO HtmlForm`)

```
mainWUI :: WuiSpec a -> a -> (a -> IO HtmlForm)
           -> IO HtmlForm
```

so that an expression (`mainWUI` $wspec\ d\ store$) evaluates to a web page containing an editor that shows the current data d and executes ($store\ d'$) when the user submits the modified data d'. The operation $store$ (also called *update form*) usually stores the modified data in a file or database, returns a web page that informs the user about the successful (or failed) modification, and proceeds with a further interaction.

The first argument of type `WuiSpec a`, also called *WUI specification*, specifies the kind of interface used to manipulate data of type `a`. This is necessary since there are usually various alternative interaction forms for identical data types. For instance, integer values can be manipulated in text fields (see last column in the table of Fig. 1) or, if the set of possible values is small, via selection boxes (see the two columns before the last one in Fig. 1). Therefore, the WUI framework provides a couple of predefined interaction forms for various data types. For instance, there are predefined entities

```
wString :: WuiSpec String
wInt    :: WuiSpec Int
```

to manipulate strings and integer values in simple text input fields, respectively. Similarly, there is an entity

```
wSelectInt :: [Int] -> WuiSpec Int
```

to select a value from a list of integers by a selection box.

In order to construct WUIs for complex data types, there are *WUI combinators* that are mappings from 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)
w4Tuple :: WuiSpec a -> WuiSpec b -> WuiSpec c ->
           WuiSpec d -> WuiSpec (a,b,c,d)
...
```

Thus,

```
wDate = wTriple (wSelectInt [1..31])
                (wSelectInt [1..12]) wInt


wPerson = w4Tuple wString wString wString wDate
```

define WUI specifications for dates and persons consisting of first name, surname, email address, and date of birth. To manipulate lists of data objects, there is a WUI combinator for list types:

```
wList :: WuiSpec a -> WuiSpec [a]
```

Thus, to manipulate lists of persons as shown in Fig.1, we apply the main construction operation `mainWUI` to the WUI specification `(wList wPerson)`, which is of type

```
WuiSpec [(String,String,String,(Int,Int,Int))] ,
```

and appropriate data values and update forms.

As discussed above, our type-oriented construction of WUIs leads to type-safe user interfaces, i.e., the user can only enter type-correct data so that the application does not need to perform any checks for this purpose. Up to now, type-correctness is interpreted w.r.t. the types of the underlying programming language. However, many applications require a more fine-grained definition of types. For instance, not every triple of natural numbers that can be entered with the WUI `wDate` above is acceptable, e.g., the triple `(29,2,1982)` is illegal from an application point of view. In order to support also correctness checks for such *application-dependent type constraints*, our framework allows to attach a computable predicate to any WUI: there is an operation (also defined as an infix operator)

```
withCondition :: WuiSpec a -> (a->Bool) -> WuiSpec a
```

that combines a WUI specification with a predicate on values of the same type so that the result specifies a WUI that accepts only values satisfying the given predicate. For instance,

```
wEmail :: WuiSpec String
wEmail =  wString `withCondition` isEmail
```

defines a WUI that accepts only syntactically correct email addresses provided that `isEmail` is a predicate on strings that is satisfied if its argument is a syntactically valid email address.

If application-specific conditions on input values are added, appropriate error messages should be provided. For this purpose, there is an operation (infix operator)

```
withError :: WuiSpec a -> String -> WuiSpec a
```

that combines a WUI specification with a specific message which is shown in case of inputs that do not satisfy the input constraints. For instance, we can improve the definition of `wEmail` with an appropriate error message as follows:

```
wEmail = wString
         `withCondition` isEmail
         `withError`     "Invalid email address:"
```

Similarly, if `correctDate` is a predicate on triples of integers that checks whether this triple represents a legal date (e.g., `correctDate (29,2,1982)` evaluates to `False`), then the WUI specification `wDate` above should be better defined by

```
wDate = wTriple (wSelectInt [1..31])
                (wSelectInt [1..12]) wInt
          `withCondition` correctDate
          `withError`     "Illegal date:"
```

Redefining the WUI for persons by

```
wPerson = w4Tuple wString wString wEmail wDate
```

the expression `(wList wPerson)` denotes a WUI specification for lists of persons that checks for valid inputs and provides the error messages shown in Fig. 1.

Furthermore, the framework to construct WUIs also supports the adaptation of WUIs for standard types to WUIs for application-specific types, the use of application-specific renderings to produce layouts different from the default layout, and the embedding of WUIs into web pages with a fixed design (headers, menu bars, etc) or the embedding of severals WUIs with separate submit buttons at arbitrary places into a single web page (avoiding any conflicts in the naming of references for input fields). The details about these features are not relevant for the contents of this paper and can be found in [21].

4. Combining WUIs and JavaScript

As mentioned in Section 1, we intend to exploit the existence of JavaScript interpreters in current web browsers to increase the functionality of WUIs. In particular, we want to transmit a JavaScript program, together with the HTML form implementing a WUI, that implements the validation of user inputs in the HTML form. With this approach, invalid inputs are detected by the web browser on the client side which provides instantaneous feedback to the user and reduces the number of client/server interactions. Note that it is not our intention to shift computations from the server side to the client side in order to reduce the load of the web server: since a web application should never trust user inputs received from a client (see Section 1), the validation of inputs by the web application is mandatory. This design decision has a number of advantages:

- It is not necessary to check all input conditions in a WUI on the client side.
- If the JavaScript program running on the client cannot compute a definite result, e.g., due to resource limitations, it causes no problem for the web application since the input is always validated on the server side.
- The same is true if JavaScript is disabled in the client's browser (e.g., due to security reasons). In this case, the web forms can still be used (in contrast to approaches that rely on JavaScript like PowerForms [7]). The only difference is that input errors are shown *after* the form has been submitted to the web server which sends back a new form with error-annotated input fields (identical to the example in Fig. 1).

In order to develop our new approach to exploit JavaScript for client-side input validation, we sketch the current WUI implementation (see [21] for more details).

The execution of a WUI by a call to `mainWUI` (see Section 3) with some WUI specification, data value, and update form is performed by the following steps:

1. The initial HTML form containing input elements to modify the given data value is generated according to the WUI specification and sent to the client's browser.
2. If the user submits the form with the modified value, the new value is extracted and the validity conditions on the various levels (since the value can be complex, there might be conditions on atomic parts as well as on constructed parts) are checked. If one of these conditions is not satisfied, a new "error answer" form with error-annotated input fields is generated and sent back to the client (see Fig. 1). Then, step 2 is repeated.
3. If all conditions on the submitted value are satisfied, the update form is applied to the new value.

Using JavaScript, we can improve step 2: If we add JavaScript code to the form sent to the client, various conditions can be checked before the modified form is sent to the web server. If one of the conditions is not satisfied, the form is not submitted but an error message is displayed by the web browser. For this case, one can distinguish two kinds of conditions:

1. Conditions on individual fields, e.g., text fields for integers, email addresses: They can be immediately checked whenever the user leaves such an input field (i.e., in case of an `onblur` event of the browser).
2. Conditions on combination of several input fields, e.g., dates represented by several select buttons: They are checked when the user clicks the submit button since it is difficult to determine a fixed point of time to check the condition before submitting the entire form.

In both cases, an error message is immediately displayed near the invalid input. In case of individual fields, the error message is shown before or above the fields, and in case of combined fields, the error message is displayed together with a frame enclosing all fields belonging to the invalid value.

In order to display the error message immediately, it must be already contained in the HTML form at the right position. This can be done by the use of style sheets that makes the error message initially invisible. For instance, the HTML form sent to the client contains the following style definitions:[3]

```
<style type="text/css">
 .hide { display: none; }
 .nohide { display: inline; color: red;
   font-weight: bold; background-color: yellow; }
</style>
```

Then the error message has initially the style class `hide` that is changed to `nohide` when the input is invalid. For instance, the

[3] The code presented here is simpler than the actual code in order to ease the understanding.

HTML code for a text field containing an email address could be as follows:

```
<span class="hide" id="MSG1">
 Invalid email address:
</span>
<input type="text" name="F1" value="smith.org"
  onblur='setErrorClassName("MSG1",
              isEmail(stringValueOf("F1")))' />
```

The function `isEmail` is the translation of the Curry function used in the WUI specification into JavaScript that will be shown below. The simple JavaScript function `setErrorClassName` sets the style class of the element identified by the first argument according to the Boolean value of the second argument (which is also returned and used in case of nested structures):

```
function setErrorClassName(m,b) {
  var errmsg = document.getElementById(m);
  if (b) { errmsg.className = "hide"; }
    else { errmsg.className = "nohide"; }
  return b;
}
```

After the introduction of the principles of combining WUIs and JavaScript, we consider the translation of Curry programs into JavaScript in the next section.

5. Translating Curry into JavaScript

An advantage of our design discussed in the previous section is the fact that it is not necessary to translate any Curry program into JavaScript. This is also not reasonable since the advanced features of a declarative multi-paradigm language like Curry (e.g., higher-order functions, laziness, logic variables, constraint solving, concurrency) requires a complex run-time system that must be transmitted to the client with the HTML form.[4]

Thus, we define a class of Curry programs that are "easy to translate" into JavaScript by omitting the more complex features of Curry.

DEFINITION 5.1. *A function f* depends on *function g if an evaluation (according to the operational semantics of Curry [1, 16]) of a call to f might evaluate a call to g. We denote by $f^* = \{f\} \cup \{g \mid f \text{ depends on } g\}$ the set of all functions on which f depends.*

Since this definition of dependency is undecidable in general, we assume in the following a decidable approximation of this property, e.g., by considering the call structure of the program rules as computed by existing analysis tools [19].

DEFINITION 5.2. *A function f is* totally defined *if it is reducible on all ground data terms (i.e., expressions without defined functions and logic variables).*

The main restrictions that we impose on functions which are translated into JavaScript are totally definedness and the exclusion of non-determinism as well as infinite data structures, i.e., we consider the functional part of Curry evaluable by an eager strategy. To characterize this class, we use some results about functional logic programs. ISX [6] ("inductively sequential programs with extra variables") denotes the class of functional logic programs where each function is defined such that its left-hand sides can be organized in a definitional tree [2] (i.e., it is inductively defined on the input types) possibly containing extra variables (variables in a rule that do not occur in its left-hand side). It is well known [3, 6] that any functional logic program (i.e., conditional constructor-based term rewriting system) can be automatically transformed into an ISX program. Therefore, we consider only ISX programs in the following.

DEFINITION 5.3. *A function f of an ISX program is* eager executable *if all functions of f^* are totally defined and their rules do not contain extra variables, and the rewrite relation generated by all rules defining f^* is terminating.*

The main motivation for this definition is the fact that eager-executable functions can be executed by an innermost rewriting strategy without changing the computed results:

PROPOSITION 5.4. *If f is an eager-executable function, then any call to f without free variables that is evaluated by innermost term rewriting produces the same result as in Curry.*

PROOF: Since we have not fully formalized all notions necessary for a detailed proof, we give only an informal sketch (the complete proof is not difficult but tedious in all its details). The requirement for ISX programs without free variables in the rules and the initial expressions implies that the evaluation strategy of Curry [1, 16] applied to such programs reduces to pure term rewriting, i.e., non-deterministic or concurrent computations do not occur. Furthermore, any innermost term rewriting strategy computes a value (i.e., a term without defined function symbols) due to totally definedness and termination (note that both requirements are necessary). Since ISX programs without extra variables are confluent, this value is unique and, thus, the same as computed by Curry. □

Note that the class of eager-executable functions is sufficient in practice. Most of the functions used as conditions in practical WUIs are either eager executable (e.g., all predicates used in the examples of this paper) or exploit advanced features that are not reasonable to translate into JavaScript (e.g., constraint solving over finite domains, see the SuDoku solver shown in [21]). If functions are not totally defined, they can be easily "totalized" by introducing particular failure values [20] so that this requirement is not a serious restriction. The requirement for the absence of extra variables is easy to check. The termination requirement is more serious due to its undecidability. However, there are many tools to check sufficient termination criteria that can be applied (e.g., AProVE [14]).[5]

The previous proposition shows that eager-executable functions can be executed by an innermost rewrite strategy, i.e., basically a call-by-value strategy that is also used in JavaScript. Thus, these functions can be more or less directly mapped into JavaScript functions. In the remaining section, we define a translation scheme for eager-executable Curry functions into JavaScript based on these considerations. Since this scheme produces a lot of code and the code size is a limiting factor of such mobile code, we discuss in the subsequent section a number of useful optimizations.

Since functions are defined in the source language Curry by pattern matching, local definitions etc, it is much more convenient to base a compiler on an intermediate language where the detailed pattern matching strategy has been already resolved, local definitions have been globalized by lambda lifting [26], and further syntactic sugar of the source language has been eliminated. This intermediate language has been also used to define the semantics of Curry [1] and various language-processing tools.

[4] A good alternative is the static integration of the Curry run-time system into the web browser, but this does not seem a viable solution.

[5] A more pragmatic approach is to ignore this requirement since web browsers usually controls the space and time requirements of executed JavaScript programs and terminate them if necessary. However, such an approach does not lead to web interfaces that might be well accepted by users.

$$
\begin{array}{rcll}
[\![r \leftarrow l]\!] & = & r \texttt{ = } \bar{l} & \text{(literal)} \\
[\![r \leftarrow x]\!] & = & r \texttt{ = } x & \text{(variable)} \\
[\![r \leftarrow c(e_1,\ldots,e_n)]\!] & = & \texttt{var } x_1\texttt{;}\, [\![x_1 \leftarrow e_1]\!]\texttt{;} \ldots \texttt{;var } x_n\texttt{;}\, [\![x_n \leftarrow e_n]\!]\texttt{;} & \text{(constructor call)} \\
 & & r \texttt{ = new Array}(c, x_1, \ldots, x_n) & \\
[\![r \leftarrow \texttt{ifThenElse}(e_1,e_2,e_3)]\!] & = & \texttt{var } x\texttt{;}\, [\![x \leftarrow e_1]\!]\texttt{; if (}x\texttt{) \{ }[\![r \leftarrow e_2]\!]\texttt{ \} else \{ }[\![r \leftarrow e_3]\!]\texttt{ \}} & \text{(conditional)} \\
[\![r \leftarrow \texttt{apply}(e_1,e_2)]\!] & = & \texttt{var } x_1\texttt{;}\, [\![x_1 \leftarrow e_1]\!]\texttt{; var } x_2\texttt{;}\, [\![x_2 \leftarrow e_2]\!]\texttt{; } r \texttt{ = } x_1(x_2) & \text{(higher-order application)} \\
[\![r \leftarrow f(e_1,\ldots,e_n)]\!] & = & \texttt{var } x_1\texttt{;}\, [\![x_1 \leftarrow e_1]\!]\texttt{;} \ldots \texttt{;var } x_n\texttt{;}\, [\![x_n \leftarrow e_n]\!]\texttt{; } r \texttt{ = } \bar{f}(x_1,\ldots,x_n) & \text{(function call)} \\
[\![r \leftarrow f(e_1,\ldots,e_{n-1})]\!] & = & \texttt{var } x_1\texttt{;}\, [\![x_1 \leftarrow e_1]\!]\texttt{;} \ldots \texttt{;var } x_{n-1}\texttt{;}\, [\![x_{n-1} \leftarrow e_{n-1}]\!]\texttt{;} & \text{(partial application)} \\
 & & r \texttt{ = function(}x\texttt{)\{ return } \bar{f}(x_1,\ldots,x_{n-1},x)\texttt{; \}} & \\
[\![r \leftarrow \textit{case } e \textit{ of } \{\overline{p_n \rightarrow e_n}\}]\!] & = & \texttt{var } x\texttt{;}\, [\![x \leftarrow e]\!]\texttt{;} & \text{(case, where} \\
 & & \texttt{swith (}x\texttt{[0]) \{} & p_i = c_i(x_{i1},\ldots,x_{in_i})\text{)} \\
 & & \quad \ldots & \\
 & & \quad \texttt{case } c_i \texttt{ : var } x_{i1} \texttt{=} x\texttt{[1]; } \ldots \texttt{;var } x_{in_i} \texttt{=} x\texttt{[}n_i\texttt{]; }[\![r \leftarrow e_i]\!]\texttt{; break ;} & \\
 & & \quad \ldots & \\
 & & \texttt{\}} &
\end{array}
$$

Figure 2. Translation of expressions into JavaScript statements

Programs of this intermediate language, also called *flat programs*, can be defined as follows, where we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$:[6]

Program
$P \quad ::= \quad D_1 \ldots D_m$

Definition
$D \quad ::= \quad f(x_1, \ldots, x_n) = e$

Expression

e	$::=$	l	(literal)
	$\mid$	x	(variable)
	$\mid$	$c(e_1, \ldots, e_n)$	(constructor call)
	$\mid$	$f(e_1, \ldots, e_n)$	(function call)
	$\mid$	*case* e *of* $\{\overline{p_n \rightarrow e_n}\}$	(case)

Pattern
$p \quad ::= \quad c(x_1, \ldots, x_n)$

A *literal* l is a number, character, or Boolean constant. Higher-order applications are not explicitly represented since we assume that they are represented by the predefined binary function `apply`. Similarly, a conditional like if-then-else is represented by a function `ifThenElse` that takes three arguments and has the following definition as a flat program:

```
ifThenElse(b,t,f) = case b of { True  → t,
                                False → f }
```

Since eager-executable functions can be automatically transformed into such flat programs [23], we define the compilation process only for such flat programs.

Due to the fact that the target JavaScript program should implement an innermost rewriting strategy, we can map each eager-executable Curry function into a definition of a JavaScript function. However, expressions in the right-hand side of function definitions cannot be mapped into JavaScript expressions since case expressions are translated into a `switch` construct which is a statement rather than an expression in JavaScript. Therefore, we map expressions into JavaScript statements that store the result value in a given variable. Hence, we define a translation function $[\![r \leftarrow e]\!]$ that translates an expression e into a statement that stores the result in variable r.

[6] We have simplified the flat language due to our requirement for eager-executable functions.

A function definition $f(x_1, \ldots, x_n) = e$ is translated into the following JavaScript function declaration:

```
function f(x1,...,xn) {
  var r;
  [[r ← e]];
  return r;
}
```

The translation of expressions is defined by a case distinction on the different kinds of expressions as shown in Fig. 2. In the translation rules, every variable x_i introduced by a `var` declaration denotes a fresh variable. In the literal translation rule, $\bar{l}$ denotes the JavaScript constant corresponding to the Curry literal l. In the function translation rule, $\bar{f}$ is f for a defined function f or the name of the corresponding JavaScript function in case of a primitive function f (e.g., arithmetic function, comparison, character conversion). Constructor applications are represented as arrays where the first element contains the constructor (e.g., in string representation or as an integer if we associate a unique index to every constructor). In the rules for constructor and function calls, we assume that c and f are fully applied (i.e., they have arity n), respectively. In a partial application, some argument in a call to an n-ary function (or constructor) f is missing (for the sake of simplicity, we define the translation scheme only for a single missing argument). In this case we exploit the higher-order features of JavaScript by generating a new function that could be later applied to the missing argument (compare translation rule for `apply`). In the case translation rule, a case distinction is performed on the current constructor (compare the translation of constructor calls) before the pattern variables x_{ij} are extracted from the data structure. An obvious optimization of this translation scheme, performed by our compiler, is the replacement of each access to a pattern variable x_{ij} in the translation of e_i by x`[`j`]`.

EXAMPLE 5.5. *Consider the following WUI specification describing an interface for a pair of integers where their sum must be positive:*

```
let posSum (x,y) = x+y > 0
 in wPair wInt wInt `withCondition` posSum
```

Using our translation scheme, the predicate `posSum` *is translated into the following JavaScript code:*

```
function posSum(x1) {
  var x2;
  var x3; x3 = x1;
  switch (x3[0]) {
    case "(,)" :    // pair constructor?
      var x4;
      var x5; x5 = x3[1];
      var x6; x6 = x3[2];
      x4 = (x5 + x6);
      var x7; x7 = 0;
      x2 = (x4 > x7); break;
  }
  return x2;
}
```

Obviously, this schematically generated code has the potential for many optimizations that should be applied in order to reduce the code size. This will be discussed in the next section.

6. Optimizations

Due to the translation of expressions into statements, nested expressions are flattened by introducing auxiliary variables. In general, this is necessary to exploit switch statements to implement pattern matching. As example 5.5 shows, this schematic translation often introduces variables that are only assigned and used once. Therefore, we introduce a code optimization that removes such variables:

Removal of single variables If there is a code sequence (i.e., with a sequential control flow between the shown points) of the form

`... var` x`; ...` x `=` e`; ...` x `...`

without any other occurrences of x, remove the declaration and assignment for x and replace the remaining single occurrence of x by e.

This specific combination of constant propagation and liveness analysis is very effective in our example:

EXAMPLE 6.1. *If we apply the removal of single variables to the generated code of Example 5.5, we obtain:*

```
function posSum(x1) {
  var x2;
  var x3; x3 = x1;
  switch (x3[0]) {
    case "(,)": x2 = (x3[1] + x3[2] > 0); break;
  }
  return x2;
}
```

The motivation to remove only variables with a single occurrence is to avoid additional computational work that could be introduced by duplicating code. Since all variables in an eager language are already evaluated, we can also remove variables with multiple occurrences that have values identical to other variables:

Removal of aliasing variables If there is a code sequence (i.e., with a sequential control flow between the declaration and assignment) of the form

`... var` x`; ...` x `=` y`; ...`

without any other assignment to x, remove the declaration and assignment for x and replace all remaining occurrences of x by y.

EXAMPLE 6.2. *If we apply the removal of aliasing variables to the code of Example 6.1, we obtain:*

```
function posSum(x1) {
  var x2;
  switch (x1[0]) {
    case "(,)": x2 = (x1[1] + x1[2] > 0); break;
  }
  return x2;
}
```

The latter code can be further improved by exploiting the fact that it is the translation of a *strongly typed* source language. Due to this property, it is not necessary to check the constructor in the switch statement, since the argument expression of the switch is always a value that belongs to a data type with a single constructor:

Removal of record constructors Let T be a type constructor with a single data constructor, i.e., defined by

`data` $T\ a_1 \ldots a_n$ `=` $C\ \tau_1 \ldots \tau_m$

Then replace "`switch (`x`[0]) {case` C`: ` s`; break;}`" by s.

EXAMPLE 6.3. *Applying the removal of record constructors to the code of Example 6.2 combined with a further removal of single variables yields the final code:*

```
function posSum(x1) {return (x1[1] + x1[2] > 0);}
```

This code is much smaller and more efficient than the originally generated code of Example 5.5 and also similar to a hand-written code.

Apart from these code optimizations, one should also consider the representation of data types in the generated JavaScript code. Primitive types like numbers or truth values have a direct correspondence between Curry and JavaScript. Constructed types are represented as array structures (see Fig. 2). An alternative representation could be objects in JavaScript, but there does not seem to be clear advantages for one these options. However, the situation is different for strings. Many input fields contain strings that should be checked according to various conditions. For instance, our small run-time system to execute JavaScript programs for WUIs contains the function `stringValueOf` that extracts a string from an input field and passes it to the corresponding check function (see the example in Section 4). Since the type of strings is `[Char]` in Curry, i.e., identical to a list of characters in order to reuse all existing polymorphic list functions also for strings, the function `stringValueOf` must convert the input string into a list of characters. However, it is well known that this representation is inefficient compared to a primitive string representation [11]. The same drawback is present here since each character in this list is represented by an (array) object whereas strings are primitive in JavaScript with a much better representation. For instance, a condition which checks whether an input field contains a non-empty string can be defined in Curry by

```
notEmpty []    = False
notEmpty (_:_) = True
```

which is translated by our techniques into

```
function notEmpty(x1) {
  var x2;
  switch (x1[0]) {
    case "[]" : x2 = false; break;
    case ":" :  x2 = true;  break;
  }
  return x2;
}
```

Since the transformation of a string into a character list where only the first element is checked could be expensive (we detected per-

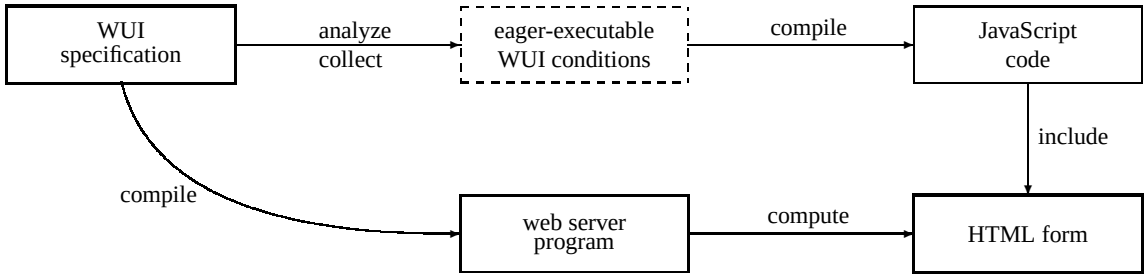

Figure 3. Structure of the WUI/JavaScript implementation

ceivable computation costs with strings containing several thousands characters), we have implemented another solution inspired by Curry's lazy evaluation strategy: the function `stringValueOf` directly returns the JavaScript string without any transformation. In order to comply with the standard list functions for processing strings in Curry, we lazily convert a string into a list if it is demanded, i.e., in front of a switch statement on lists. Thus, the code for the function `notEmpty` is modified as

```
function notEmpty(x1) {
  var x2;
  if ((typeof(x1) == "string")) {
    x1 = string2charlist(x1);
  }
  switch (x1[0]) {
    case "[]" : x2 = false; break;
    case ":" :  x2 = true;  break;
  }
  return x2;
}
```

where the function `string2charlist` converts an empty string into the empty list `[]` and a non-empty string into a non-empty list consisting of the first character and the remaining string (in place of the tail list).

7. Implementation

This section sketches the implementation of our framework that is visualized in Fig. 3. The complete implementation is available with the current distribution of PAKCS [22].

As discussed in detail in Section 5, it is not intended to translate all possible conditions in WUIs into corresponding JavaScript code. Therefore, one has to select the conditions satisfying the criteria presented in Section 5 before the translation starts. Thus, the compilation of a web page containing WUIs is performed by the following steps:

1. The source program is analyzed and eager-executable conditions in WUIs are marked.
2. The eager-executable conditions are collected together with the functions on which they depend (which might be distributed in various imported modules) into a single Curry program.
3. This Curry program is translated into a JavaScript program following the methods described in Sections 5 and 6.
4. The original Curry program is compiled with a standard Curry compiler and installed on the web server.
5. If a client demands a web page containing the WUI, the Curry program computes the corresponding HTML form and sends it to the client together with the generated JavaScript program, as described in Section 4.

Program	Curry		JS		JSO	
	lines	bytes	lines	bytes	lines	bytes
posSum	1	21	19	278	3	63
isEmail	27	654	183	3050	77	1598
person	71	1624	404	6546	126	2777
exam	102	2333	629	10072	211	4613

Table 1. Code size of some programs

Note that the implementation of WUIs, as described in [21], must also consider the integration of JavaScript code in the HTML form. For this purpose, the individual functions to generate HTML code from WUI specifications (they are implicitly contained in the WUI specifications but not directly accessible) generate also the calls to the JavaScript code for eager-executable conditions. These calls are attached to input fields if possible (in case of text fields with `onblur` events) and also collected for the complete WUI and attached to the submit button. The check of the complete WUI follows an innermost strategy in case of hierarchical data structures (like list of persons containing dates): first, the basic inputs parts are checked and, if they do not contain an error, the parts constructed from these parts are checked. This strategy is reasonable since it avoids superfluous error messages related to global properties if the individual parts contain input errors.

Our current implementation that comes with PAKCS does not integrate an analyzer for eager-executable conditions, i.e., the first step of the implementation must currently be done by the programmer (the connection of a termination analysis tool for Curry is part of future work). For this purpose, the WUI programmer can mark a condition to be translated into JavaScript by using the WUI function `withConditionJS` instead of `withCondition`.[7] All the remaining steps of the implementation are fully automatic and do not require any help by the programmer.

In order to provide an impression of the size of the generated JavaScript code, Table 1 contains the results of compiling some example programs from Curry into JavaScript. The columns contain the sizes of the source Curry program (including all dependent functions but without comments), the generated JavaScript code without optimization (JS), and the generated JavaScript code including the optimization presented in Section 6 (JSO). For each class of programs, the number of code lines and the code size in bytes is shown. The first three programs are WUI conditions mentioned in this paper, and program `exam` consists of the conditions of a web-based examination management tool. The difference be-

[7] The explicit marking of conditions that should be translated into JavaScript is also useful in the presence of an analyzer for eager-executable functions, since there could be situations where eager-executable conditions should not be transferred to the client, e.g., if they contain confidential algorithms.

tween the entries in the columns JS and JSO clearly shows that the optimizations of Section 6 are important and effective.

8. Related Work

Web-based user interfaces are important for many modern applications. In principle, a dynamic web page can be implemented in any programming language since the requirements on CGI programs that generate dynamic web pages are very low due to the text-based CGI protocol. Although scripting languages like Perl or PHP are quite common for this purpose, they lack support for reliable programming (e.g., types, static checking of declarations) so that various approaches to implement web interfaces with higher-level programming languages have been developed. Some of them are discussed in the following.

One of the early domain-specific languages for web programming is MAWL [27]. It supports the checking of well-formedness of HTML documents by writing HTML documents with some gaps that are filled by the server before sending the document to the client. Since these gaps are filled only with simple values, the generation of documents whose structure depend on complex data is largely restricted. More complex tree structures are supported in DynDoc [33] (part of the `<bigwig>` project [8]) which supports higher-order document templates, i.e., the gaps in a document can be filled with other documents that can also contain gaps. In order to validate user inputs in HTML forms, the `<bigwig>` project proposes PowerForms [7], an extension of HTML with a declarative specification language to annotate acceptable form inputs. Since the specification language is based on regular expressions, it is less powerful than our approach which supports any computable predicate on inputs. Similarly to our approach, PowerForms are translated into JavaScript so that input checking is done on the client side. As a drawback, the implementation of PowerForms completely relies on JavaScript so that they cannot be used if JavaScript is disabled. Finally, the `<bigwig>` project is based on a domain-specific language for writing dynamic web services while we exploit the features of the existing high-level language Curry.

Similar to the basic HTML programming library of Curry [17], there are also libraries to support HTML programming in other functional and logic languages. For instance, the PiLLoW library [9] is an HTML/CGI library for Prolog. Since Prolog is not strongly typed, static checks on the form of HTML documents are not directly supported. Furthermore, there is no higher-level support for complex interaction sequences as required in typical user interfaces.

Meijer [29] developed a CGI library for Haskell that defines a data type for HTML expressions together with a wrapper function that translates such expressions into a textual HTML representation. However, it does not offer any abstraction for programming sequences of interactions (e.g., by event handlers). These must be implemented in the traditional way by choosing strings for identifying input fields, passing states as hidden input fields etc. Thiemann [34] proposed a representation of HTML documents in Haskell that ensures the well-formedness of documents by exploiting Haskell's type class system. In [35] he extended this approach by combining it with the ideas of [17] to implement interaction sequences by an event handler model. Although his approach also supports typed input fields similarly to our WUIs, it is more restricted. It does not support arbitrary conditions on input data or type-based combinators for input fields, and the layout of the generated web pages is more restrictive (due to the monadic implementation in the functional base language).

The iData toolkit [31, 32] is a framework, implemented with generic programming techniques in the functional language Clean, to construct type-safe web interfaces. Similarly to WUIs, editors for typed values are created in a type-oriented way. However, in contrast to our framework, the editable data elements are identified by strings that might cause consistency problems similarly to scripting languages like Perl or PHP. Furthermore, apart from well-typedness, validity conditions on input data as in our approach are not supported.

The Google Web Toolkit (GWT[8]) is a framework to implement dynamic web pages in Java. GWT contains a compiler to translate the developed Java programs into a set of JavaScript and HTML files. Similarly to our approach, GWT proposes the use of a statically typed language to catch many programming errors at compile time instead of programming in JavaScript. In contrast to our proposal, GWT has no specific support to construct type-safe web forms in a high-level manner.

Ruby on Rails[9] is a framework to generate web interfaces to manipulate data stored in databases from the corresponding database schema. Similarly to our WUIs, Rails reduces the code that must be explicitly written by generating most of it from the database schema. Rails is based on the dynamically-typed, object-oriented language Ruby, whereas we have exploited the strong typing discipline of Curry.

9. Conclusion

We have proposed a new framework to construct web-based user interfaces in a declarative way that is combined with features of JavaScript in order to exploit existing technologies without efforts for the application programmer. The construction of WUIs is type-oriented, i.e., the programmer selects basic WUI components and combine them with specific combinators in order to obtain a WUI that can be applied to manipulate the data of the application domain. An important feature of WUIs is the possibility to include computable conditions on input data. Since these conditions are checked before the data is transferred to the application program, the application must only specify such conditions but need not check their validity or implement the necessary interactions with the user to correct wrong inputs.

In this paper we have shown how to exploit the existing technology for client-side input checking within this framework but without additional efforts for the application programmer. For this purpose, we have characterized a class of functions that can be easily translated into a language like JavaScript with a call-by-value semantics. Conditions in WUIs that are implemented by functions belonging to this class are automatically translated into corresponding JavaScript code. This code is used on the client side when the user manipulates or submits the application data in a web form generated by the WUI description.

The advantages of programming with WUIs has been already shown in a previous paper [21] and various applications based on this concept. The advantage of the current work is the reuse of the technology available in almost every web browser without efforts for the programmer. Instead of programming in different languages, e.g., writing scripts in a dynamically typed, interpreted language like JavaScript, which often leads to unreliable programs and must be kept in conformance with the application (remember that server-side input checking is always necessary to avoid security risks), we propose to generate the necessary code from the existing WUI specification. In principle, this concept can be also applied to other programming languages than Curry. However, it has been demonstrated that the combined features of a multi-paradigm language

[8] `http://code.google.com/webtoolkit/`

[9] `http://www.rubyonrails.com`

like Curry can be exploited to provide better APIs for such libraries [17, 21].

For future work it would be interesting to provide useful tools for the automatic termination or complexity analysis. They can be used for a better characterization of programs that should be translated into JavaScript. For instance, functions with a high computational complexity should not be executed on the client's browser, even if they are terminating. Furthermore, the compilation into JavaScript could be improved. Although the current translation into recursive JavaScript functions is sufficient in our practical applications, future applications might demand for better compilation schemes, e.g., tail recursion optimization. Finally, one could exploit the more recent Ajax framework to increase the interaction, e.g., by executing complex conditions on the web browser before submitting the form, or by extending the framework to allow conditions of type "`IO Bool`", i.e., which depend on the server's state and must be executed on the web server during the user interaction with his web browser.

Acknowledgments

The author is grateful to the anonymous referees for their suggestions and to Bernd Braßel, Sebastian Fischer, and Frank Huch for their constructive comments on this work.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.

[2] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.

[3] S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.

[4] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[5] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.

[6] S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pp. 87–101. Springer LNCS 4079, 2006.

[7] C. Brabrand, A. Møller, M. Ricky, and M.I. Schwartzbach. PowerForms: Declarative Client-side Form Field Validation. *World Wide Web Journal*, Vol. 3, No. 4, pp. 205–214, 2000.

[8] C. Brabrand, A. Møller, and M.I. Schwartzbach. The `<bigwig>` Project. *ACM Transactions on Internet Technology*, Vol. 2, No. 2, pp. 79–114, 2002.

[9] D. Cabeza and M. Hermenegildo. Internet and WWW Programming using Computational Logic Systems. In *Workshop on Logic Programming and the Internet*, 1996. See also `http://clip.dia.fi.upm.es/Software/pillow/`.

[10] J. Correas, J.M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 104–119. Springer LNCS 3057, 2004.

[11] D. Coutts, D. Stewart, and R. Leshchinsky. Rewriting Haskell Strings. In *Proc. 9th International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, pp. 50–64. Springer LNCS 4354, 2007.

[12] S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.

[13] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 5th edition edition, 2006.

[14] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, pp. 281–286. Springer LNCS 4130, 2006.

[15] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[16] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

[17] 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)*, pp. 76–92. Springer LNCS 1990, 2001.

[18] M. Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, Vol. 2004, No. 5, 2004.

[19] M. Hanus. CurryBrowser: A Generic Analysis Environment for Curry Programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pp. 61–74, 2006.

[20] M. Hanus. Reporting Failures in Functional Logic Programs. In *Proc. of the 15th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*, pp. 49–62, 2006.

[21] 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)*, pp. 27–38. ACM Press, 2006.

[22] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2007.

[23] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.

[24] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.

[25] S.H. Huseby. *Innocent Code: A Security Wake-Up Call for Web Programmers*. Wiley, 2003.

[26] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Functions. In *Functional Programming Languages and Computer Architecture*, pp. 190–203. Springer LNCS 201, 1985.

[27] D.A. Ladd and J.C. Ramming. Programming the Web: An Application-Oriented Language for Hypermedia Services. In *4th International World Wide Web Conference*, 1995.

[28] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.

[29] E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, Vol. 10, No. 1, pp. 1–18, 2000.

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

[31] R. Plasmeijer and P. Achten. The Implementation of iData - A Case Study in Generic Programming. In *Proc. of the 17th International Workshop on Implementation and Application of Functional Languages (IFL 2005)*. Trinity College, University of Dublin, Technical Report TCD-CS-2005-60, 2005.

[32] 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)*, pp. 242–258. Springer LNCS 3945, 2006.

[33] 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*, pp. 290–301, 2000.

[34] P. Thiemann. Modelling HTML in Haskell. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 263–277. Springer LNCS 1753, 2000.

[35] 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)*, pp. 192–208. Springer LNCS 2257, 2002.

[36] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.