Type-Oriented Construction of Web User Interfaces *

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

Abstract

This paper proposes a new technique for the high-level construction of type-safe web-oriented user interfaces. Our approach is useful to equip applications processing structured data with interfaces to manipulate these data in an efficient and maintainable way. The interfaces are web-based, i.e., the data can be manipulated with standard web browsers without any specific requirements on the client side. In order to support type-safe user interfaces, i.e., interfaces where users can only input type-correct data (types can be standard types of a programming language as well as any computable predicate on the data), we propose a set of type-oriented building blocks from which interfaces for more complex types can be easily constructed. This technique leads to a very concise and maintainable implementation of web-based user interfaces.

We show an implementation of this concept in the declarative multi-paradigm language Curry. In particular, its 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.1.6 [*Programming Techniques*]: Logic 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; H.5.2 [*Information Interfaces and Presentation*]: User Interfaces

General Terms Languages

Keywords Functional Logic Programming, User Interfaces, Web Programming

1. Motivation

The construction of user interfaces for applications manipulating structured data is usually a complex and often tedious task. Our experiences from various applications show that in many cases the effort to implement a user interface is equal or even bigger than the implementation of the application itself, in particular, if the application is implemented in a declarative programming language which offers a high level of abstraction for application programming (e.g., see [14] or the collection of Curry applications[1]). Thus, there is a demand to support the efficient construction of maintainable user interfaces. Scripting languages with toolkits and libraries, like Tcl/Tk, Perl, or PHP, are one approach to support this goal. However, 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 higher-order functions) so that they are often used to implement the user interface whereas the application logic is implemented in some other language. It is well known that such combinations could cause security leaks in web applications [16]. Therefore, we prefer to implement the user interface with the programming language of the application logic by providing specific libraries for this purpose. In this paper we present such an approach for the case of declarative programming and web user interfaces (WUIs) where the client uses a standard web browser for communicating with the application. We will see that the decrease in flexibility of low-level approaches is negligible in comparison to the increase of programming efficiency and reliability supported by an integration into a higher-level language.

Our approach is useful in situations where a web-based editor should be constructed for data of an application program, i.e., the user should be provided with an HTML form to manipulate some data of the application. For this purpose we assume that the application program supplies the WUI with the current data of the application and an operation to store the modified data. It is obvious that this is not a restriction since application programs usually have such a functionality. Using our concept, nothing more is required to construct WUIs in a high-level way by a few lines of program code. Our programming model can be characterized by the following features:

- The construction of a WUI is *type-oriented*, i.e., the definition of a WUI follows the structure of the data types of the application.
- There is a set of *basic WUIs* to manipulate data of basic types, e.g., integers, truth values, strings, finite sets. This set can be easily extended since there is a clear methodology to implement such basic WUIs.
- There is 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.
- It is ensured that an update of the data is only performed with *type-correct inputs*. If the user tries to input illegal data (e.g., incorrect integer constants), the WUI does not accept the data and ask the user to correct the input. Thus, the application pro-

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

[1] `http://www.informatik.uni-kiel.de/~curry/applications.html`

gram need not check the data and perform appropriate actions (e.g., providing error forms to correct the input etc).

- Type-correct inputs (in the sense of types used in programming languages) are often not sufficient in real applications. For instance, strings containing email addresses must have a particular form, a date like "February 29, 2006" is illegal, or two input fields containing a password and the repeated password must be always identical. For this purpose, WUIs can be *restricted with any computable predicate* so that input data is only accepted if it satisfies the specified predicate. Furthermore, WUIs can be customized to provide *application-specific error messages* in case of illegal inputs.
- WUIs can be adapted to other data types in order to provide a simple method to define *WUIs for user-defined data types*. For instance, there exist WUI combinators for tuples that can be easily adapted to a user-defined record type by mapping tuples to records. Although this method is often sufficient to construct WUIs for user-defined types, there is also a methodology to extend the standard set of WUI combinators with new application specific combinators.

In principle, our ideas can be implemented in various programming languages. However, in order to support a compact, high-level, and type-safe implementation, some requirements to the underlying programming language are necessary. Therefore, we provide a concrete implementation of our concept in the declarative multi-paradigm language Curry [10, 15]. As we will see, the integration of functions as first-order objects, logic variables, and strong typing is exploited in our implementation.

In the next section, we review some concepts of Curry and functional logic programming in order to understand the rest of the paper. HTML programming in Curry is reviewed in Section 3. Section 4 introduces the ideas of WUIs and their type-oriented construction. Section 5 sketches the implementation of our approach. Section 6 discusses related work before we conclude in Section 7.

2. Functional Logic Languages and Curry

Modern functional logic languages [9] integrate important features of functional and logic languages in order to provide a variety of programming concepts. For instance, the concepts of demand-driven evaluation and higher-order functions from functional programming are combined with logic programming features like computing with partial information (logic variables), unification, and nondeterministic search for solutions. This combination, supported by optimal evaluation strategies [1] and new design patterns [2], leads to better abstractions in application programs, e.g., as shown for programming dynamic web pages [11]. The declarative multi-paradigm Curry [10, 15] is a functional logic language extended by concurrent programming concepts and has been used in various applications. In the following, we review those elements of Curry that are necessary to understand the subsequent paper. More details about Curry's computation model and a complete description of all language features can be found in [10, 15].

Curry is a multi-paradigm declarative language that combines in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). 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 [20], 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 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. Concurrent programming is supported by primitives to suspend computations and evaluate constraints concurrently.

EXAMPLE 1. *The following Curry program defines the data types of Boolean values, possible values (*`Maybe`*), union types (*`Either`*), polymorphic lists, and functions to compute the concatenation of lists and the last element of a list:*

```
infixr 5 ++

data Bool       = True    | False
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
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
```

For instance, `[]` *(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.*[2] *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 general, functions are defined by (*conditional*) *rules* of the form

$$f\ t_1 \ldots t_n\ \texttt{|}\ c\ \texttt{=}\ e\ \ \texttt{where}\ vs\ \texttt{free}$$

with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$. The condition and the `where` parts can be omitted if c and vs are empty, respectively. The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable.

A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2,

[2] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

i.e., this expression is evaluated by proving both argument constraints concurrently. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable patterns. However, specific Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [13].

The operational semantics of Curry, described in detail in [10, 15], is based on an optimal evaluation strategy [1] 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 [1] for details). Curry also offers standard features of functional languages, like higher-order functions (e.g., "\\x `->` e" denotes an anonymous function that assigns to each x the value of e) or monadic I/O (which is identical to Haskell's I/O concept [25]).

3. HTML Programming in Curry

This section surveys the model supported in Curry for programming dynamic web pages. This model exploits the functional and logic features of Curry and is available through the library `HTML` which is part of the PAKCS distribution [13]. The ideas of this library and its implementation are described in detail in [11].

If one wants to write a program that generates an HTML document, one must decide about the representation of such documents inside a program. 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 [7, 19, 23]. For instance, the type of HTML expressions is defined in Curry as follows:

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

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 (because there are a few HTML elements without a closing tag, like `<hr>` or `<br>`, there is also the constructor `HtmElem` to represent these elements).

Note that this definition of HTML documents covers only their tree-like structure but does not put further restrictions on the documents so that combinations not conform with the HTML standard can be created. Although this can be avoided with refined definitions and more sophisticated type systems [23], we use this definition for the sake of simplicity since such details are not important for our subsequent approach.

Writing HTML documents in the form of this data type might be tedious. Therefore, the `HTML` library defines several functions as useful abbreviations (`htmlQuote` transforms characters with a special meaning in HTML into their HTML quoted form):

```
htxt   s     = HtmlText   (htmlQuote s)
h1     hexps = HtmlStruct "h1" [] hexps
bold   hexps = HtmlStruct "b"  [] hexps
breakline    = HtmlElem   "br" []
...
```

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. As before, there are also useful abbreviations:

```
form title hexps = HtmlForm title [] hexps

standardForm title hexps =
             form title (h1 [htxt title] : hexps)
```

The intention of a dynamic web page is to represent some information that depends on the environment of the web server (e.g., stored in data bases). Therefore, a dynamic web page has always the type "`IO HtmlForm`", i.e., it is an I/O action that retrieves some information from the environment and produces a web document.

Dynamic web pages become more interesting by processing user inputs during the generation of a page. For this purpose, HTML provides various input elements (e.g., text fields, text areas, check boxes). A subtle point in HTML programming is the question how the values typed in by the user are transmitted to the program generating the answer page. This is the purpose of the Common Gateway Interface (CGI) but the details are completely hidden by the `HTML` library. The programming model supported by the `HTML` library 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 to compute the answer document. In order to access the user input, the event handler is a function from a "CGI environment" (holding the user input) into an I/O action that returns an HTML document. A *CGI environment* is simply a mapping from CGI references, which identify particular input fields, into strings. Thus, there are the following type synonyms:

```
type CgiEnv      = CgiRef -> String
type HtmlHandler = CgiEnv -> IO HtmlForm
```

Thus, a submit button takes string (shown in the button) and an event handler as a parameter, where the associated event handler is called with the appropriate CGI environment when the user pushes the submit button:

```
button :: String -> HtmlHandler -> HtmlExp
```

An input element that occurs in an HTML form has a parameter of type `CgiRef` so that an event handler can refer to its contents. For instance, a text input field has the following type:

```
textfield :: CgiRef -> String -> HtmlExp
```

The first argument is its reference and the second argument is the initial contents of this field.

What are the elements of type `CgiRef`, i.e., the *CGI references* to identify input fields? In traditional web programming (e.g., raw CGI, Perl, PHP), one uses strings to refer to input elements. However, this has the risk of programming errors due to typos and does not support abstraction facilities for composing documents (see [11] for a more detailed discussion). Here, free variables are useful. Since it is not necessary to know the concrete representation of a CGI reference, the type `CgiRef` is abstract. Thus, we use free variables as CGI references. Since each input element has a CGI reference as a parameter, the free variables of type `CgiRef` are the links to connect the input elements with the use of their contents in the event handlers. For instance, the following program implements a (dangerous!) web page where a client can submit a file name. As a result, the contents of this file (stored on the web server) are shown in the answer document:[3]

```
getFile = return $ form "Question"
             [htxt "Enter local file name:",
              textfield fileref "",
              button "Get file!" handler]
 where
  fileref free

  handler env = do
    contents <- readFile (env fileref)
    return $ form "Answer"
      [h1 [htxt ("Contents of " ++ env fileref)],
       verbatim contents]
```

Since the locally defined name `fileref` (of type `CgiRef`) is visible in the right-hand side of the definition of `getFile` as well as in the definition of `handler`, it is not necessary to pass it explicitly to the event handler. Note the simplicity of retrieving values entered into the form: since the event handlers are called with the appropriate CGI environment containing these values, they can easily access these values by applying the environment to the appropriate CGI reference, like `(env fileref)`.

This structure of CGI programming is made possible by the functional as well as logic programming features of Curry. Although some aspects of this programming model, in particular, the use of event handlers, can also be implemented in a purely functional language [24], other aspects like the use of free variables for locally defined input fields can only be simulated in a restricted way, e.g., by the use of monads to create unique references, since the local creation of globally unique identifiers is a typical functional logic design pattern [2]. More details and the advantages of this programming model are discussed in [11].

4. Constructing Web User Interfaces

In this section we present our concept to construct WUIs from a programmer's point of view, i.e., we present the WUI API before we discuss its implementation in the next section.

4.1 Basic WUIs

In order to support the construction of WUIs for a large class of applications, we only require that the application program supplies the WUI with the current state of the data and an operation to store the data modified by the user. Thus, the main operation to construct a WUI could have the type signature (remember that dynamic web pages are of type `IO HtmlForm`)

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

so that an expression of the form (`mainWUI` 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. Therefore, the result type of $store$ is `IO HtmlForm` rather than `IO ()`.

[3] The predefined right-associative infix operator f `$` e denotes the application of f to the argument e.

Note that `mainWUI` is polymorphic in the type of the data to be processed so that it can be applied to any kind of data. However, in practice one needs HTML forms that depend on the data types and exploit the various possibilities of HTML input elements (strings, text areas, radio buttons, selection boxes, etc). Therefore, `mainWUI` should be better treated as an overloaded function. Although type classes as in Haskell [20, 26] allow an elegant formulation of overloaded operations, this is not sufficient for our purpose since there could be different interaction forms for identical data types. For instance, strings can be modified with simple text input fields or with larger text areas, finite sets of constants can be represented as radio buttons or selection boxes, etc. Based on these considerations, `mainWUI` should also take a specification of the interaction form as a further argument so that we obtain the type signature

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

Here, `WuiSpec a` is a data type to specify an interaction form, which we also call *WUI specification* or *widget*, to modify values of type `a`. For instance, to edit simple strings, there is a predefined entity

```
wString :: WuiSpec String
```

defining a WUI element that shows the string in a simple text input field. For larger text paragraphs, there is an entity

```
wTextArea :: (Int,Int) -> WuiSpec String
```

defining a WUI element that shows the string in a text area with a particular height and width (the argument pair of this entity). As an example for another data type, there is an entity

```
wInt :: WuiSpec Int
```

defining a WUI element that shows an integer in a text input field. Note that WUI elements are type safe, i.e., if the user inputs a non-integer in such an input field, the implementation of the WUI emits an error message and asks the user to correct the input. As an example, consider the definition of a form that just shows its argument:[4]

```
resultForm :: a -> IO HtmlForm
resultForm v =
 return $ form "Result"
               [htxt ("Modified value: "++show v)]
```

Then the expression

```
mainWUI wInt 42 resultForm
```

evaluates to a simple web page with an input field (with initial value `42`, see left part of Figure 1) where integer values can be modified. If the user tries to modify the input field with a non-integer value, an error page is returned that allows the user to modify the input with a correct value (see right part of Figure 1).

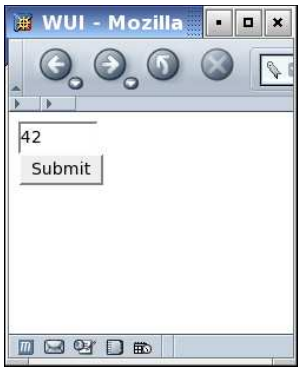
Similarly, there are WUI elements for other elementary data types. For instance,

```
wSelect :: (a->String) -> [a] -> WuiSpec a
```

defines a WUI element to select a value from a list of values by a selection box. The first argument is a function to show an element as a string (shown in the selection box) and the second argument contains the list of elements to be selected. As a concrete example, we can derive the following WUI element to select Boolean values (where the parameters are the strings that are shown for the values `True` and `False` in the selection box, respectively):

```
wSelectBool :: String -> String -> WuiSpec Bool
```

[4] `show` is a universal operation to portray any value as a string.



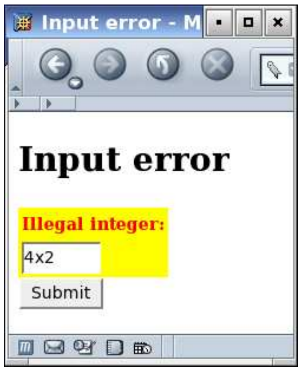


Figure 1. A simple integer WUI (left) with an input error (right)

```
wSelectBool t f = wSelect (\b -> if b then t else f)
                          [True,False]
```

If one provides an interface to manipulate larger data structures, it is often required that some parts of the structure (e.g., unique identifiers like insurance numbers) should only be shown but not modified. For this purpose, we also provide a WUI element

```
wConstant :: (a->HtmlExp) -> WuiSpec a
```

which requires a function to visualize the (non-editable) value by an HTML expression. Similarly, there is a WUI element

```
wHidden :: WuiSpec a
```

that does not visualize the value in the interface. This is useful to hide parts of larger structures that should not be shown in the interface (e.g., internal identifiers such as database keys). The construction of WUIs for such larger structures is the topic of the following section.

4.2 WUI Combinators

Similarly to the use of type constructors for the construction of complex data types from simpler types, WUIs for complex types can be constructed from WUIs for simpler types by *WUI combinators*. A WUI combinator is a mapping 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)
...
```

Thus, the expression

```
mainWUI (wTriple wString wInt
                 (wSelectBool "male" "female"))
        ("Joe",30,True) resultForm
```

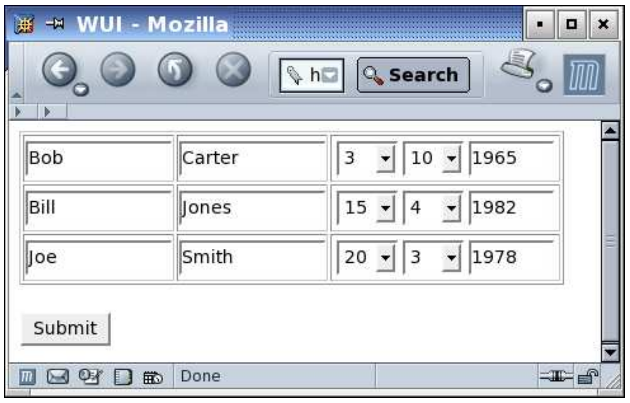
evaluates to an HTML form to edit a triple consisting of a string, an integer, and a Boolean.

In order to manipulate lists of data objects, there is a WUI combinator for list types:

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

For instance, consider the following derived WUI types for dates and persons consisting of first name, surname, and date of birth:

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



Bob	Carter	3	10	1965
Bill	Jones	15	4	1982
Joe	Smith	20	3	1978

Figure 2. A WUI for a personal data list

```
wPerson = wTriple wString wString wDate
```

In order to manipulate lists of persons, we can use the WUI (`wList wPerson`) which is of type

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

Figure 2 shows an example form of this type. Note that the elements in a list are vertically aligned in a table in the HTML form. Later we will see how such default renderings can be easily modified.

In order to construct WUIs for union types, there is a combinator for `Either` types:

```
wEither :: WuiSpec a -> WuiSpec b
           -> WuiSpec (Either a b)
```

Other types with more alternatives can be easily reduced to a combination of several `Either` types. This is sufficient in practice since we will also provide possibilities to adapt WUIs for standard types to arbitrary user-defined types (see below).

4.3 Constraining WUIs

As already discussed, 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 `(29,2,2006)` is illegal from an application point of view. In order to support also correctness checks for such *application-dependent type constraints*, one can attach a computable predicate to any WUI. For this purpose, 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.

As a simple example, we can derive a WUI that accepts only non-empty strings:

```
wRequiredString :: WuiSpec String
wRequiredString =
           wString `withCondition` (not . null)
```

Consider a predicate `isEmail` on strings that is satisfied if its argument is a syntactically valid email address. Then we can easily define a WUI that accepts only correct email addresses:

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

Obviously, one can also add conditions on non-elementary WUIs. For instance, if `correctDate` is a predicate on triples of integers that checks whether this triple represents a legal date (e.g., `correctDate (29,2,2006)` evaluates to `False`), then the WUI specification `wDate` above should be better defined by

```
wDate = wTriple (wSelect show [1..31])
                (wSelect show [1..12])
                wInt
          ‘withCondition‘ correctDate
```

If specific conditions on input values are added, appropriate error messages should be provided. For this purpose, there is an operation (also defined as an 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 above WUI `wEmail` with an appropriate error message by the following improved definition:

```
wEmail = wString
          ‘withCondition‘ isEmail
          ‘withError‘     "Illegal email address:"
```

4.4 Adapting WUIs

In Section 4.2 we have shown how to define WUIs for complex data types constructed by standard type constructors like tuples or lists. WUIs for application-specific types can be implemented by the following techniques:

- One implements WUI combinators for the application-specific types using the same techniques as for the standard WUI combinators. As we will see in Section 5, this is not difficult although technically non-trivial and tedious to implement in detail.
- One uses the standard WUI combinators and adapts them to application-specific types by defining mappings between standard types and application-specific types. Since such mappings are not difficult to define in a functional logic language, we propose this method in practice.

In order to support the second alternative, there is an operation to transform WUIs from one type to another:

```
transformWSpec :: (a->b, b->a) -> WuiSpec a
                               -> WuiSpec b
```

The corresponding value mappings are provided as parameters to this transformation. Note that value mappings must be given for both directions: first, a value of type `b` must be transformed into a value of type `a` in order to manipulate it with the WUI for `a`-values, and after submitting this (modified) value, it must be transformed into a `b`-value in order to pass it to the application. Fortunately, if both value types are isomorphic, it is sufficient to provide only one transformation as a (bijective) function since the other direction can be automatically derived exploiting the functional *logic* features of Curry. For instance, consider the following data types for dates and persons:

```
data Date   = Date Int Int Int
data Person = Person String String Date
```

Then one can easily define a function that maps values of the tuple-oriented person type used in Section 4.2 for the WUI `wPerson` into `Person` values:

```
tuple2person :: (String,String,(Int,Int,Int))
                -> Person
tuple2person (first,name,(d,m,y)) =
             Person first name (Date d m y)
```

Now, exploiting the concept of function patterns [3], where defined functions can also occur at pattern positions, we can easily define the inverse of `tuple2person` by

```
person2tuple (tuple2person t) = t
```

so that WUIs for values of type `Person` can be specified as follows:

```
wPersonType :: WuiSpec Person
wPersonType =
transformWSpec (tuple2person,person2tuple) wPerson
```

It is not difficult to provide a general definition of this technique so that we can define a “WUI adapter” where only one bijective mapping need to be provided:

```
adaptWSpec :: (a->b) -> WuiSpec a -> WuiSpec b
adaptWSpec a2b = transformWSpec (a2b,invert a2b)
```

Here, `invert` is a function that inverts a given function by the concept of function patterns above, i.e., it is required that the first argument to `adaptWSpec` is a bijective function like `tuple2person` (of course, this property is undecidable so that it must be ensured by the programmer), otherwise the programmer must provide an explicit definition of the inverse mapping and use `transformWSpec`.

4.5 Application-specific Rendering

To support the efficient construction of WUIs, basic WUIs and WUI combinators have standard renderings that produce reasonable layouts in most cases. For instance, `wSelect` shows the alternative elements in a selection box, `wList` aligns the renderings of the individual list elements vertically in a table, or tuple combinators combine the elements horizontally (see Figure2). Nevertheless, it is sometimes desirable to have a particular layout in some applications. For this purpose, our approach also support the modification of such standard renderings that is described in the following.

Conceptually, we define a *rendering* as a function that combines the layout of the substructures of some WUI (i.e., the HTML expressions for the component WUIs in a WUI combinator) into a new combined layout specified by some HTML expression. Thus, a rendering is a function of the following type:

```
type Rendering = [HtmlExp] -> HtmlExp
```

For instance, the standard renderings for lists and tuples are defined as follows:

```
renderList  hexps = table (map (\h->[[h]]) hexps)
renderTuple hexps = table [map (\h->[h]) hexps]
```

`renderList` maps the elements into a table where each element is put into one row, and `renderTuple` maps the elements into a table consisting of one row where each element is put into a column of that row.

As mentioned above, each WUI element has a *default rendering*, like `renderTuple` for WUIs containing tuples or `renderList` for list WUIs. Since the main rendering of basic WUIs is fixed (e.g., selection boxes for `wSelect` or text input fields for `wString` or `wInt`), the rendering function for such basic WUIs is simply `head`, i.e., the projection on the first element of the list of HTML expressions. Thus, each WUI element contains a default rendering function of type `Rendering`. This default rendering can be changed so that one can decorate basic WUIs with further HTML structures or define new alignments or combinations for combined WUIs.

To change the default rendering of a WUI, there is an operation to transform a WUI into a WUI with a new rendering (also defined as an infix operator):

```
withRendering :: WuiSpec a -> Rendering -> WuiSpec a
```

Thus, we can easily derive from `wList` a new WUI combinator for lists where the elements are horizontally aligned in a table:

```
wHList :: WuiSpec a -> WuiSpec [a]
wHList wspec = wList wspec
                `withRendering` renderTuple
```

We can combine the combinators `wList` and `wHList` to a combinator for matrices, i.e., lists of lists of elements visualized as a matrix:

```
wMatrix :: WuiSpec a -> WuiSpec [[a]]
wMatrix wspec = wList (wHList wspec)
```

The latter combinator can be exploited to define a WUI for "SuDoku" puzzles (see Section 4.7) consisting of a 9×9 matrix of digits (where zero is shown as a blank, see upper part of Figure 3):

```
wSudoku :: WuiSpec [[Int]]
wSudoku =
  wMatrix (wSelect (\i -> if i==0 then " "
                                   else show i)
                   [0..9])
```

Note that this definition of a matrix WUI works well only for matrices where the cells in each column have the same width, since each row of the matrix is a separate table. This drawback can be easily avoided by a new definition of `renderList` that merges a "table of tables" (i.e., a table where each row contains a table) into a single table. Actually, this is the implementation of `renderList` in our library, otherwise a WUI for list of tuples, like in Figure 2, would not look well.

4.6 Embedding WUIs in Web Pages

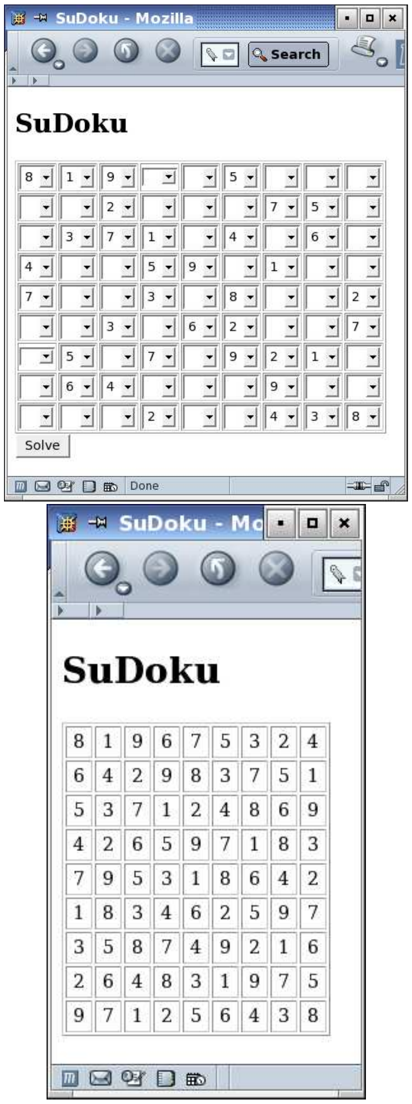
As we have seen in Section 4.1, we can generate web pages containing a form for a particular WUI specification and data values by a call to the function `mainWUI` (see Figure 1). However, in most applications one wants to embed the manipulation of data into web pages with a fixed design (headers, menu bars, etc). For this purpose, the translation of a WUI into a complete form is not desirable so that it is reasonable to provide a function

```
wui2html :: WuiSpec a -> a -> (a -> IO HtmlForm)
            -> (HtmlExp,HtmlHandler)
```

that is similar to `mainWUI` but returns an HTML expression implementing the WUI layout and a handler that can be attached to a submit button. Using this function, WUIs can be integrated into any web page and the function `mainWUI` can be implemented as follows:

```
mainWUI wuispec val store =
 let (hexp,handler) = wui2html wuispec val store
  in return $ form "WUI" [hexp, breakline,
                          button "Submit" handler]
```

Moreover, our WUI library also offers functions to embed a WUI into a web page having "holes" for the WUI layout and handler. Note that the underlying library for HTML programming [11] is compositional: a web form (i.e., an expression of type `HtmlForm`, see Section 3) can contain HTML expressions with an arbitrary number of submit buttons for different handlers. Thus, one can also embed severals WUIs with separate submit buttons at arbitrary



SuDoku

8	1	9			5			
		2				7	5	
	3	7	1		4		6	
4			5	9		1		
7			3		8			2
		3		6	2			7
	5		7		9	2	1	
	6	4				9		
			2			4	3	8

SuDoku

8	1	9	6	7	5	3	2	4
6	4	2	9	8	3	7	5	1
5	3	7	1	2	4	8	6	9
4	2	6	5	9	7	1	8	3
7	9	5	3	1	8	6	4	2
1	8	3	4	6	2	5	9	7
3	5	8	7	4	9	2	1	6
2	6	4	8	3	1	9	7	5
9	7	1	2	5	6	4	3	8

Figure 3. A WUI for "SuDoku" puzzles (upper part: input, lower part: solution output)

places in a single web page without any conflicts in the naming of references for input fields.

4.7 Summary

Before discussing the implementation of our WUI concept, which we have introduced in this section from a programmer's point of view, we summarize its basic features and provide a complete example.

Our approach is tailored towards an efficient implementation of WUIs where the application programmer is freed from implementing the process of retrieving the input data from individual input fields or checking the type correctness or other constraints on input values. Although some default design decisions support the straightforward construction of WUIs, our concept also allows the modification of these defaults. For instance, the programmer can define parts of the data as constant (i.e., non-editable) or hide it in the form. Standard renderings can be easily modified or complex input conditions can be attached to WUIs. Furthermore, WUIs for user-defined data types can be easily derived by providing bijective mappings to standard data types like lists, tuples, or unions.

During the design of our concept, we have also investigated possibilities to provide standard WUIs for dynamic data structures, such as lists, with interaction elements to extend them (e.g., add a new value to a list of values). However, our framework does not provide specific support for this due to the numerous possibilites to extend such structures (e.g., insert elements at arbitrary positions in a list, extend tree structures at arbitrary nodes or leaves). Nevertheless, it is fairly easy to implement HTML forms that support such value extensions at well-defined places, e.g., by generating WUIs with default or blank input values.

As a small example for the construction of a complete web application exploiting our concept, we show the implementation of a web interface to a "SuDoku" solver. A SuDoku puzzle consists of a 9×9 matrix of digits between 1 and 9 so that each row, each column, and each of the nine 3×3 sub-matrices contain pairwise different digits. The challenge is to find the missing digits if some digits are given (see upper part of Figure 3). We assume that a solver exists as an operation

```
solveSudoku :: [[Int]] -> IO (Maybe [[Int]])
```

that takes a matrix of integers (where a zero value represents an unknown digit) and returns a solution or `Nothing` if no solution exists (an implementation of this operation using finite domain constraints in Curry is shown in the appendix).

The initial web form has a matrix (e.g., a distinct example) as input and allows the user to change some digits by the WUI `wSudoku` defined above. It puts the HTML expression and handler into a standard form with a "Solve" button:

```
initForm s =
 let (hexp,handler) = wui2html wSudoku s solveForm
  in return $ standardForm "SuDoku"
                  [hexp, button "Solve" handler]
```

The integer matrix defined by the user in this web form is processed by the operation `solveForm` that calls the SuDoku solver and returns a web page that shows the solution (see lower part of Figure 3):

```
solveForm m = do
  sol <- solveSudoku m
  return $ standardForm "SuDoku"
   (if sol==Nothing
    then [h1 [htxt "No solution"]]
    else [fst (wui2html wMatConst
                    (fromJust sol) initForm)])
 where wMatConst = wMatrix (wConstant (htxt . show))
```

Note that we also use our WUI concept to show the (non-editable) constant value of a solution. In this case, the button and continuation store form is not used. Therefore, we put `initForm` at the corresponding argument position.

Now we can provide an initial "empty" form by evaluating the expression

```
initForm (map (const (take 9 (repeat 0))) [1..9])
```

which can be interpreted as the main function of our web script. This example shows the compactness of code that is necessary to attach a web user interface to an existing application.

5. Implementation

This section sketches the implementation of our WUI concept using the features for HTML programming [11] of Curry. The complete implementation is available with the current distribution of PAKCS [13].

As apparent from the type of the function `wui2html` (see Section 4.6), the implementation must be able to translate a WUI specification together with a value and an update form into an HTML expression that implements the WUI layout and a handler for a submit button. Furthermore, WUI combinators take the functionality of the component WUIs as input and produce the functionality of the combined WUI. Therefore, a WUI specification must contain some information about translating WUIs into appropriate HTML expressions and generating the corresponding event handlers. Based on these considerations, one could define the type `WuiSpec` as follows:

```
data WuiSpec a = WuiSpec (a -> HtmlState)
                         (CgiEnv -> WuiState -> a)

type HtmlState = (HtmlExp,WuiState)
```

Thus, a WUI specification consists of two operations. From the current value to be manipulated by the user, the first operation produces the corresponding HTML expression contained in the web form together with a state that is passed (by the implementation of `wui2html`) to the event handler (such pairs are of type `HtmlState`). The state is necessary to store the references of the input fields of the HTML expression in order to enable the event handler to extract the corresponding user inputs. Consequently, the second operation of a WUI specification returns the value modified by the user in the form from the current CGI environment and the state.

For instance, assume operations to encode and decode a CGI reference in a state (of type `WuiState`, see below):

```
cgiRef2state :: CgiRef -> WuiState

state2cgiRef :: WuiState -> CgiRef
```

Then one can implement the basic WUI for strings as follows:

```
wString :: WuiSpec String
wString =
 WuiSpec
  (\v -> let ref free in
         (textfield ref v, cgiRef2state ref))
  (\env state -> env (state2cgiRef state))
```

In order to implement a WUI combinator like `wPair`, it is necessary to compose and decompose several states into one. For this purpose, we assume the existence of the operations

```
states2state :: [WuiState] -> WuiState

state2states :: WuiState -> [WuiState]
```

so that one can implement the pair combinator as follows:

```
wPair :: WuiSpec a -> WuiSpec b -> WuiSpec (a,b)
wPair (WuiSpec showa reada) (WuiSpec showb readb) =
 WuiSpec
  (\(va,vb) -> let (hexpa,statea) = showa va
                   (hexpb,stateb) = showb vb
     in (renderTuple [hexpa,hexpb],
         states2state [statea,stateb]))
  (\env state ->
        let [statea,stateb] = state2states state
         in (reada env statea, readb env stateb))
```

Other WUI combinators can be constructed in a similar way. It is also not difficult to implement the transformation of WUIs between data types (see Section 4.4) by applying the transformation functions at appropriate places:

```
transformWSpec :: (a->b,b->a) -> WuiSpec a
                               -> WuiSpec b
transformWSpec (a2b,b2a) (WuiSpec showa reada) =
  WuiSpec (\vb -> showa (b2a vb))
          (\env state -> a2b (reada env state))
```

The data type of states used in WUIs can be defined as follows:

```
data WuiState = Ref      CgiRef
              | CompNode [WuiState]
              | AltNode  (Int,WuiState)
              | Hidden   String
```

The constructor `Ref` represents a reference to an elementary input field (see `cgiRef2state` and `state2cgiRef` above), `CompNode` represents a state combined from several states (see `states2state` and `state2states` above), `AltNode` represents alternatives, i.e., the union of components as used in a combinator like `wEither`, and `Hidden` is used to keep a string representation of hidden values (used to implement `wConstant` and `wHidden`).

Unfortunately, the implementation shown so far is not able to support the complete functionality of WUIs presented in Section 4. For instance, the functionality of WUIs can be modified by operations like `withRendering`, `withError`, or `withCondition` so that WUIs must also contain information about the rendering, error messages in case of input errors, and predicates on the input data. For this purpose, we can define the following data type to keep this information in one structure:

```
type WuiParams a = (Rendering, String, a->Bool)
```

These parameters must be stored in a WUI specification (in order to modify them by an operation like `withRendering`) and passed to the WUI operations so that we obtain the following improved definition of the type of WUI specifications:

```
data WuiSpec a =
  WuiSpec (WuiParams a)
          (WuiParams a -> a -> HtmlState)
          (WuiParams a -> CgiEnv -> WuiState -> a)
```

However, one final modification of this type is necessary in order to implement the reaction to illegal inputs in a form. In this case, the input values are not returned to the update form but they are shown in a new HTML form together with the corresponding error message (see right of Figure 1). Therefore, the return type of the second WUI operation must be modified to a `Maybe` value (where `Nothing` is returned in case of an illegal input) together with an HTML expression and state for the subsequent error form. Hence, we obtain our final definition of the type of WUI specifications:

```
data WuiSpec a =
   WuiSpec (WuiParams a)
           (WuiParams a -> a -> HtmlState)
           (WuiParams a -> CgiEnv -> WuiState
                        -> (Maybe a, HtmlState))
```

Note that the structure of the error form must be always constructed, i.e., also in case of valid inputs, since illegal inputs can occur at any level of the construction of the output value.

According to this final definition of the type `WuiSpec`, the implementation of the basic WUI for strings is as follows:

```
wString :: WuiSpec String
wString =
 WuiSpec
  (head, "?", const True)
  (\(rnd,_,_) v -> stringWidget rnd v)
  (\wparams env s ->
         checkLegalInput wparams stringWidget
                         (env (state2cgiRef s)))
 where
  stringWidget render v =
    let ref free in
    (render [textfield ref v], cgiRef2state ref)
```

The polymorphic function `checkLegalInput` is responsible to check a user input and returns, according to the validity of the input, a corresponding `Maybe` value together with a form that is used if the input is invalid:

```
checkLegalInput :: WuiParams a
                -> (Rendering -> a -> HtmlState)
                -> a
                -> (Maybe a,HtmlState)
checkLegalInput (rnd,errmsg,legal) v2widget v =
 if legal v
 then (Just v,  v2widget rnd v)
 else (Nothing, v2widget (renderError rnd errmsg) v)
```

The function `renderError` is of type

```
renderError :: Rendering -> String -> Rendering
```

and combines a given rendering with an error message so that the resulting rendering produces an HTML expression containing the input rendering surrounded by the error message (see right part of Figure 1).

In a similar way, the implementation of WUI combinators, like `wPair` shown above, can be adapted to the final definition of `WuiSpec` with error handling.

A tricky detail of the complete implementation of our WUI concept is the generation of input forms and error handling forms. This is done by the main operation `wui2html` introduced in Section 4.6. This operation translates a WUI specification, a value, and an update form into an HTML expression and a handler that implements the edit form. Basically, an expression

```
(wui2html wuispec val store)                    (1)
```

where the argument `wuispec` has the form

```
WuiSpec wparams showhtml readval
```

is evaluated as follows:

1. Compute the initial HTML form containing the input elements to modify the given value `val`: this is done by evaluating the expression `(showhtml wparams val)`. The result is a pair `(hexp,wstate)` of type `HtmlState`, where `hexp` is the HTML expression specifying the layout of the initial form and `wstate` is a `WuiState` value containing the references to the input fields of `hexp`.
2. The value `hexp` is the first component of the result of (1).
3. The second component (the event handler) of the result of (1) is computed as follows. First of all, the lambda abstraction

   ```
   \env -> readval wparams env wstate
   ```

   has the functional type `CgiEnv -> (Maybe a, HtmlState)`. In order to obtain the required event handler from this abstraction, i.e., a function of type `HtmlHandler`, we have to process the result `(mb,hst)` of the body expression “`readval wparams env wstate`”:

   - If `mb` has the form `Just` v, then the modified value v is legal so that we return `(store` v`)` which is of the required type `IO HtmlForm`.

- If `mb` is the value `Nothing`, then the current user input is not legal. In this case we use `hst = (errhexp,errwstate)` which consists of an HTML expression `errhexp` specifying the layout of the error handling form and a state `errwstate` containing the references to the input fields of `errhexp`. Then our event handler proceeds with step (2.) where it uses `errhexp` and `errwstate` instead of `hexp` and `wstate`, respectively.

This case distinction together with the recursion in the second case implements the dynamic generation of correct error handling forms. Note that, conceptually, the error handling forms are always constructed, i.e., even in the case of legal user inputs. However, thanks to lazy evaluation, the concrete HTML expressions are only computed when they are really demanded.

With these type definitions and explanations, the implementation of the concrete code can be done without difficulties. Thus, we omit further details here. The interested reader might look into the source code of the `WUI` library that can be found in the current PAKCS distribution [13].

6. Related Work

The implementation of web-based user interfaces has an increasing relevance in modern applications. In principle, dynamic web pages 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.

MAWL [18] is an early domain-specific language for programming web interfaces. 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 depends on complex data is largely restricted. To relax this restriction, MAWL offers special iteration gaps which can be filled with list values. More complex tree structures are supported in DynDoc [22] (part of the `<bigwig>` project [5]) 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 [4], 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. Furthermore, PowerForms are translated into JavaScript so that input checking is done on the client side. This has the advantage to reduce network traffic but the disadvantage that such forms cannot be used on clients where JavaScript is disabled for security reasons. 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 approach for HTML programming in Curry [11], there are also libraries to support HTML programming in other functional and logic languages. For instance, the PiLLoW library [7] is an HTML/CGI library for Prolog. Due to the untyped nature of Prolog, static checks on the form of HTML documents are not supported. Furthermore, there is no higher-level support for complex interaction sequences as required in typical user interfaces.

Meijer [19] has 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 [23] proposed a representation of HTML documents in Haskell that ensures the well-formedness of documents by exploiting Haskell's type class system. In [24] he extended this approach by combining it with the ideas of [11] to implement interaction sequences by an event handler model. Although his approach also supports typed input fields similarly to our concept, it is more restricted. It does not support arbitrary conditions on input data or type-based combinators for input fields. Furthermore, the layout of the generated web pages is more restrictive. Since Thiemann uses a purely functional language and a monadic model to create references to input fields, submit buttons must always be placed below the input fields. Such a restriction is not required in our approach due to the use of logic programming features (free variables as references).

Plasmeijer and Achten [21] proposed the iData toolkit to implement type-safe web interfaces in the functional language Clean. Similarly to our approach, editors for typed values are created in a type-oriented way. However, there are also important differences. For instance, the editable data elements are identified in the program by strings so that it is the task of the programmer to use unambiguous names throughout the program for different data. The use of strings, that are not checked at compile-time, is a source of potential programming errors similarly to scripting languages like Perl or PHP. Furthermore, the application using the iData toolkit must be built around the iData model since it provides specific support for making the considered data persistent. This is in contrast to our approach which has no specific restriction on the application. For instance, persistent data can be stored in files or in databases using standard access methods [8, 12], and a web-based user interface can be added in an independent way after the application logic has been implemented.

7. Conclusion

We have presented a new concept to implement web user interfaces in a type-oriented way. As we have shown in the paper, the type-oriented construction leads to a high-level, compact, and maintainable implementation of the interface. The construction is based on basic WUI components for elementary data types and powerful combinators to construct WUIs for complex data types. Furthermore, our approach has several advantages:

- Interfaces are type safe, i.e., it is ensured that the modified values satisfy the given types or constraints.
- Interfaces can be integrated into separately designed web pages. There are no restrictions on the layout, e.g., the submit buttons can be put at any place in the web form.
- Interfaces are compositional, i.e., forms can contain any number of value editors and submit buttons where it is ensured that conflicts between the references of the different input fields do not occur.
- The editing facilities are separated from the application as in the classical model/view/controller paradigm for user interfaces [17]: by defining several WUI specifications for the same data type, one can have different views on the same data.
- Interfaces have various defaults to produce reasonable results without much effort. However, it is easily possible to mod-

ify these defaults (e.g., renderings, input conditions, error messages) for specific applications.

- Existing interfaces can be transformed into interfaces for application-specific types by providing mappings between the data types.

Due to these advantages, our WUI concept is already used in several applications. One simple application, a solver for SuDoku puzzles, has been shown in this paper. The compactness of the code (12 lines of code for the solver and 9 lines of code for the web interface) shows the advantages of functional logic languages and declarative programming. It is questionable whether such a compactness can be obtained in other universal programming languages.

For future work it would be interesting to adapt parts of this programming model to other programming languages. Furthermore, the checking of user inputs could be improved for specific cases. Currently, the validity of user inputs is checked on the server side. This has the advantage that no specific requirements are placed on client side (e.g., enabled JavaScript) and one could use any predicate implemented in the base language Curry to specify the validity of user inputs. However, the disadvantage is an increased network traffic in case of illegal user inputs. Instantaneous feedback on incorrect inputs on the client side could be provided by JavaScript if it is enabled on the client side. Thus, it would be interesting to translate simple predicates into JavaScript programs that are sent with the generated forms to the client, e.g., as done with PowerForms [4]. However, this does not free the server-side application from input checking in order to avoid security risks from hand-crafted malicious client inputs.

Acknowledgments

The author is grateful to the anonymous referees for their suggestions to improve the paper and to Sebastian Fischer for helpful comments and suggestions to improve the implementation of the WUI library.

References

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

[2] 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.

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

[4] 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.

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

[6] B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, Vol. 2004, No. 6, 2004.

[7] 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/`.

[8] 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.

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

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

[11] 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.

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

[13] 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/`, 2006.

[14] M. Hanus and F. Huch. An Open System to Support Web-based Learning. In *Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pp. 269–282. Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, 2003.

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

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

[17] G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface in Smalltalk-80. *Journal of Object-Oriented Programming*, Vol. 1, No. 3, pp. 26–49, 1988.

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL'89*, pp. 60–76, 1989.

A. A SuDoku Solver

The following program implements a SuDoku solver in Curry using constraints over finite domains [13]. The SuDoku puzzle is represented as a matrix of integers between 1 and 9 containing free variables for unknown digits:

```
sudoku :: [[Int]] -> Success
sudoku m =
 domain (concat m) 1 9 &
 foldr1 (&) (map allDifferent m) &
 foldr1 (&) (map allDifferent (transpose m)) &
 foldr1 (&) (map allDifferent (squaresOfNine m)) &
 labeling [FirstFailConstrained] (concat m)

-- translate a matrix into a list of elements
-- of small 3x3 squares
squaresOfNine :: [[a]] -> [[a]]
squaresOfNine [] = []
squaresOfNine (l1:l2:l3:ls) =
  group3Rows [l1,l2,l3] ++ squaresOfNine ls
 where group3Rows l123 = if null (head l123) then []
             else concatMap (take 3) l123 :
                  group3Rows (map (drop 3) l123)
```

The definition of the constraint `sudoku` is straightforward. First, the domain of all matrix elements is defined to be between 1 and 9. Then, the constraints for different elements (`allDifferent`) in rows, columns, and 3×3 squares are established by higher-order functions and standard operations on lists before concrete solutions are tested by a first-fail labeling strategy.

A solution to a `sudoku` constraint is computed by the standard operator `getOneSolution` for encapsulating search (see [6] for a discussion on encapsulating search in Curry) that returns `Nothing` if no solution exists for a constraint abstraction, or one solution s in the form `Just` s. The auxiliary operation `transDigit` translates zeros from the input matrix into free variables.

```
solveSudoku :: [[Int]] -> IO (Maybe [[Int]])
solveSudoku s =
  getOneSolution (\m -> m =:= map (map transDigit) s
                      &> sudoku m)
 where
  transDigit i = if i==0 then let x free in x else i
```