Declarative Programming of User Interfaces*

Michael Hanus and Christof Kluß

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

Abstract. This paper proposes a declarative description of user interfaces that abstracts from low-level implementation details. In particular, the user interfaces specified in our framework are executable as graphical user interfaces for desktop applications as well as web user interfaces via standard web browsers. Thus, our approach combines the advantages of existing user interface technologies in a flexible way without demands on the programmer's side. We sketch an implementation of this concept in the declarative multi-paradigm programming language Curry and show how the integrated functional and logic features of Curry are exploited to enable a high-level implementation of this concept.

1 Motivation

The implementation of a good user interface for application programs is a necessary but often non-trivial and tedious task. In order to support programmers in the implementation of user interfaces, one can find specific libraries that reflect different approaches to the construction of user interfaces. From a user's perspective, there are two kinds of user interfaces (UIs) that are currently the most important ones on conventional desktop computers:

Graphical User Interfaces (GUIs): These are user interfaces that followed the early textual user interfaces on single host computers. GUIs enabled non-expert users to easily interact with application programs. They provide a good reaction time (since they run on the local host) and are relatively easy to install as any other program, i.e., usually they are distributed with the executable of the application program. On the negative side, application programs with GUIs require some installation efforts if many users want to use them on their desktops, because one has to install them on all desktops that might have different configurations or operating systems. Moreover, they are difficult to maintain during their life time since updates must be performed on all existing installations.

Web User Interfaces (WUIs): These are user interfaces that became popular with the world-wide web and its opportunities for user interaction via dynamic web pages. In this case, the application runs on a web server and the user interacts with the application via a standard web browser. Thus, applications with WUIs are relatively easy to install for many users since every single user needs only a web browser on

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

A. Gill and T. Swift (Eds.): PADL 2009, LNCS 5418, pp. 16–30, 2009.
© Springer-Verlag Berlin Heidelberg 2009

his local host (which is usually already installed). Moreover, such applications are easy to maintain since one has to update the central installation on the web server only. On the negative side, WUIs have a moderate reaction time (since the web server is contacted for every state-changing interaction) and a complete application is more difficult to install on a single host (since one has to install and configure a web server).

A few years ago, there was also another important difference between GUIs and WUIs: the model of interaction. In application with GUIs, the user could immediately change the content of many widgets by mouse events, whereas with WUIs, each page containing user input has to be sent to the web server which returns a new web page with some modified content. However, this disadvantage of WUIs has been decreased or omitted by the development of the Ajax framework that supports an interaction with a web server without submitting and receiving complete new pages from the web server [7].

From these considerations, it is reasonable to combine the advantages of both kinds of user interfaces in a single framework so that the programmer has no additional burden to select between GUIs or WUIs (or both) for his application. This paper presents a concrete proposal of such a concept and its implementation in the declarative multi-paradigm language Curry.

In the following section, we review the main features of functional logic programming and Curry as required in this paper. Section 3 describes the concepts of our framework followed by a few examples shown in Section 4. Implementation issues and extensions are sketched in Sections 5 and 6 before we conclude in Section 7 with a discussion of related work.

2 Functional Logic Programming and Curry

In this section we review the basic concepts of functional logic programming with Curry that are relevant for this paper. More details can be found in a recent survey on functional logic programming [13] and in the definition of Curry [17].

Functional logic languages integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. Modern languages of this kind [8,17,19] 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 [1] and new design patterns [2], leads to better abstractions in application programs, e.g., as shown for programming with databases [3,6] or web programming [10,12,14]. The declarative multi-paradigm language Curry [8,17] is a functional logic language extended by features for concurrent programming. In the following, we review the elements of Curry that are relevant to understand the contents of this paper. Further features (e.g., constraints, search strategies, concurrency, declarative I/O, modules), more details about Curry's computation model, and a complete description of the language can be found in [17].

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 [22], 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-class citizens and 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 *rules* of the form "$f\ t_1 \ldots t_n \mid c = e$" 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 rule can be applied if its left-hand side matches the current call and its condition, if present, is satisfiable.

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

```
data Bool     = True    | False
data Maybe a = Nothing | Just a
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 concatenation function "`++`" is written 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]

As one can see in this example, logic programming is supported by admitting function calls with free variables (see "`ys ++ [x]`" above) and constraints in the condition of a defining rule. For instance, 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`).

A *constraint* is any expression of the built-in type `Success`. For instance, an *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifi-

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

able constructor terms. Specific Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints (e.g., PAKCS [15]).

The operational semantics of Curry, described in detail in [8,17], is based on an optimal evaluation strategy [1] which is a conservative extension of lazy functional programming and (concurrent) logic programming. Curry also offers standard features of functional languages, like higher-order functions, modules, or monadic I/O (which is identical to Haskell's I/O concept [27]). Thus, "`IO` α" denotes the type of an I/O action that returns values of type α. For instance, the predefined I/O action `getChar` has the type "`IO Char`", i.e., it returns the next character from the keyboard when it is applied. Similarly, the predefined I/O action `readFile` has the type "`String -> IO String`", i.e., it takes a string (the name of a file) and returns the contents of the file when it is applied.

3 Specifying User Interfaces

In this section we describe our proposal for the declarative programming of user interfaces that can be executed either on a local host as a GUI (e.g., by the use of Tcl/Tk [21]) or as a WUI on a web server that is accessed by a standard web browser.

In order to develop appropriate abstractions for high-level UI programming, one has to analyze the essential components of these programming tasks. Based on earlier work on programming GUIs and WUIs with functional logic languages [9,10,12], one can distinguish the following ingredients of UI programming:

Structure: Each UI has a specific hierarchical structure which typically consists of basic elements (also called *widgets*), like text input fields or selection boxes, and composed elements, like rows or columns of widgets. Thus, UIs have a tree-like structure which can be easily specified by an algebraic data type in a declarative language.

Functionality: If the user interacts with UI elements by mouse or keyboard clicks, these UI elements emit some events on which the application program should react. A convenient way to connect the application program to such events is the concept of *event handlers*, i.e., functions that are associated to events of some widget and that are called whenever such an event occurs. Usually, the event handlers use the functionality of the application program to compute some data that is shown in the widgets of the UI. Thus, event handlers are associated to some widgets but need to refer to other widgets independently of the structural hierarchy. This means that UIs have not only a hierarchical (layout) structure but also a logical (graph-like) structure that connects the event handlers with various widgets of the UI structure. In previous works on GUI and WUI programming [9,10] it has been shown that free (logic) variables are an appropriate feature to describe this logical structure and to avoid many problems that occur if fixed strings are used as references to UI elements as in traditional GUI programming (e.g., [21,26]) or WUI programming (e.g., [4,20]).

Layout: In order to support a visually appealing appearance of a UI, it should be possible to influence the standard layout of a UI. Whereas in older approaches layout

and structural information are often mixed (e.g., as in Tcl/Tk or older versions of HTML, and similarly in previous approaches to declarative GUI/WUI programming [9,10]), it has been realized that these issues should be distinguished in order to obtain clearer and reusable implementations. For instance, current versions of HTML recommend the use of cascading style sheets (CSS) to separate structure from layout.

The distinction between structure, functionality, and layout and their appropriate modelling in a declarative programming language are the key ingredients to our framework for UI programming. Although parts of these ideas can be found in our previous works [9,10,12], our current novel approach abstracts more from the underlying technology (Tcl/Tk, HTML/CGI) so that it enables a common method to specify user interfaces. In the following, we propose a concrete description of the structure, functionality, and layout of UIs in the language Curry by presenting appropriate data types and operations on them. In principle, one can transfer these ideas also to other declarative languages (where some restrictions might be necessary). However, we will see that the combined functional and logic programming features of Curry are exploited for our high-level and application-oriented description of UIs.

As already discussed, UIs have a hierarchical structure that can be appropriately described by the following data type:

```
data UIWidget = Widget WidgetKind       -- kind of widget
                       (Maybe String)   -- possible contents
                       (Maybe UIRef)    -- possible reference
                       [Handler]        -- event handlers
                       [StyleClass]     -- layout elements
                       [UIWidget]       -- subwidgets
```

In order to avoid unnecessary restrictions, the definition of a widget is quite general. In principle, one could also enumerate all kinds of widgets and distinguish between widgets having no structure (basic widgets) and widgets with structure (e.g., rows, columns). For the sake of generality, we have chosen one widget constructor where the concrete kind of widget is given as the first component (of type `WidgetKind`). The last two components are a list of layout elements (see below) and the widgets contained in this widget, respectively. The second component contains the possible contents of the widget (e.g., the entry string of a text input field, `Nothing` for widget combinators like row or column), the third component a possible reference to a widget used by other event handlers, and the fourth component a list of handlers for the various events that can occur in this widget. Concrete examples for widgets are shown below after we have discussed the other data types used in widgets.

Event handlers need to refer to other widgets independently of the widget hierarchy. Therefore, a widget can be equipped with an identity used as a reference by event handlers. Many approaches to user interface programming, like Tcl/Tk or HTML/CGI, use string constants as identifiers. Such approaches are error prone since a typo in a string constant causes a run-time error which is usually not detected at compile time. In order to provide a more reliable approach, we adapt the idea of previous works on declarative GUI and WUI programming [9,10] and make the type of widget references

abstract. Thus, one cannot construct "wrong" identifiers but has to use free variables (whose declarations are checked at compile time) for this purpose. Therefore, our `UI` library contains a type declaration

```
data UIRef = ...
```

where only the type name `UIRef` but no data constructor is exported, i.e., `UIRef` is an abstract type. Since no constructor of this data type is available to the user of the `UI` library, the only reasonable way to use values of type `UIRef` is with a free variable (see below for a concrete example).

In general, event handlers are used for two main purposes. Either they should perform some calculations and show their results in some specific widgets of the UI, i.e., they influence the state of the UI, or they should change the state of the underlying application program, e.g., the execution of an event handler might change some application data that is stored in a file or database. In order to support the latter functionality, the result type of an event handler is always "`IO ()`", i.e., an event handler might have a side effect on the external world. Since there are also I/O actions to influence the state of the UI (see below), this result type of event handlers ensures that event handlers can influence the state of the UI as well as the state of the application program.

Furthermore, the calculations and actions performed by event handlers usually depend on the user inputs stored in the widgets of the interface, i.e., these input values must be passed as parameters to the event handlers. This can be adequately modelled by an *environment* parameter that is conceptually a mapping from widget references to the string values stored in the widgets. In order to abstract from the concrete implementation of such environments, our `UI` library contains the type declaration

```
data UIEnv = ...
```

where only the type name `UIEnv` is exported. Moreover, the `UI` library contains the type declarations

```
data Command = Cmd (UIEnv -> IO ())

data Handler = Handler Event Command
```

where `Event` is the type of possible events issued by user interfaces:

```
data Event = DefaultEvent | FocusIn | FocusOut
           | MouseButton1 | MouseButton2 | MouseButton3
           | KeyPress | Return | Change  | DoubleClick
```

Therefore, each element in the list of event handlers of a widget specifies a command (an I/O action depending on the value of some environment) that is executed whenever the associated event occurs.

The type `WidgetKind` specifies the different kinds of widgets supported by our library. Some constructors of this type are

```
data WidgetKind = Col | Row | Label | Button | Entry
                | TextEdit Int Int | ...
```

The constructors `Col` and `Row` specify combinations of widgets as columns and rows, respectively. `Label` is a widget containing a string not modifiable by the user, `Button`

is a simple button, `Entry` is an entry field for a line of text, and `TextEdit` is a widget to edit larger text areas (the parameters are the height and width of the edit area).

Since it is tedious to define all widgets of a user interface by using the constructor `Widget` only, the library contains a number of useful abbreviations, like

```
col ws        = Widget Col   Nothing    Nothing    [] [] ws
row ws        = Widget Row   Nothing    Nothing    [] [] ws

label str     = Widget Label (Just str) Nothing    [] [] []

entry ref str = Widget Entry (Just str) (Just ref) [] [] []

button cmd label =
  Widget Button (Just label) Nothing
                [Handler DefaultEvent (Cmd cmd)]) [] []
```

For instance, a simple UI showing the text "`Hello World!`" and a button to exit the UI can be specified as follows:

```
col [label "Hello World!",
     button exitUI "Stop"]
```

`exitUI` is a predefined event handler to terminate the UI. The environment passed to event handlers can be accessed and modified by the predefined I/O actions `getValue` and `setValue` that take a widget reference as their first argument. Thus, "`getValue r e`" returns the value of the widget referenced by `r` w.r.t. environment `e`, and "`setValue r v e`" updates the value of the widget referenced by `r` so that it becomes visible to the user.

In order to influence the layout of UIs, widgets can take a list of style parameters of type `StyleClass`. This type contains options to align the widget or the text contained in it, set the font and color of the widget's text, set the background color, and so on. The styles of a widget can be dynamically changed by predefined operations like `setStyles`, `addStyles`, etc.

4 Examples

In order to demonstrate the concrete application of our concept, we show a few programming examples in this section. As a first example, consider a simple counter UI shown in Fig. 1. Using our library, its structure and functionality is specified as follows:

```
counterUI = col [label "A simple counter:",
                 entry val "0",
                 row [button incr   "Increment",
                      button reset  "Reset",
                      button exitUI "Stop" ]]
 where
   val free

   reset env = setValue val "0" env

   incr  env = do v <- getValue val env
                  setValue val (show (readInt v + 1)) env
```

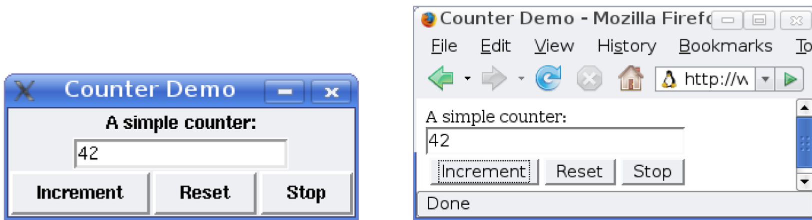

Fig. 1. A simple counter UI executed as a GUI (left) and as a WUI (right)

The free variable `val` (of type `UIRef`) denotes the reference to the entry field containing the string representation of the counter's value. It is used by the event handler `reset` to set the value of this entry widget to `"0"`. The event handler `incr` reads the current value of this widget (by "`getValue val env`") before replacing it by its incremented value (since the values in the widgets are strings, the string is transformed into an integer by "`readInt v`").

The UI specification can be executed by the predefined I/O action `runUI` that takes a string (usually shown as the label of the window containing the UI) and a UI specification as parameters. For instance, the counter UI shown above is executed by evaluating the main expression

```
runUI "Counter Demo" counterUI
```

Many interactive applications contain a state which is shown and modified by a UI. We want to demonstrate the implementation of such kinds of UIs with our concept by a simple desk calculator UI shown in Fig. 2. The implementation of this UI requires the access of the UI to some state that can be modified by the event handlers associated to the different buttons. In our application, the value of the state is a pair `(d,f)` containing the current operand `d` and an accumulator function `f` that is applied to `d` when the button "=" is pressed (this idea is due to [26]). In order to allow the change of the state's value by any event handler of the calculator UI, we model the calculator's state with `IORefs`, a concept from Haskell to deal with mutable state. `IORefs` are references to stateful objects, where their states can only be accessed and changed by the predefined I/O actions `readIORef` and `writeIORef` (in order to ensure referential transparency). Thus, the calculator UI can be implemented as follows (where the parameter `stref` of type `IORef (Int,Int->Int)` is an `IORef` to the calculator's state):

```
calcUI stref =
    col [entryS [Class [Bg Yellow]] display "0",
         row (map cbutton ['1','2','3','+']),
         row (map cbutton ['4','5','6','-']),
         row (map cbutton ['7','8','9','*']),
         row (map cbutton ['C','0','=','/'])]
  where
    display free

    cbutton c = button (buttonPressed c) [c]
```

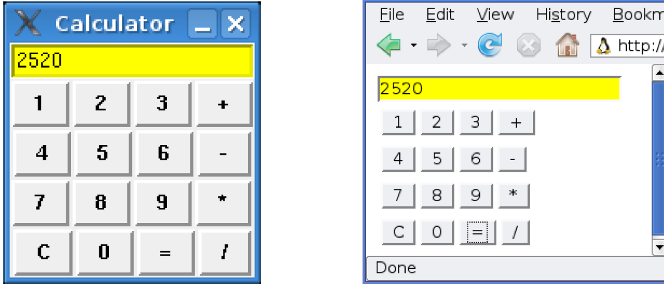

Fig. 2. A simple desk calculator UI executed as a GUI (left) and as a WUI (right)

```
buttonPressed c env = do
   state <- readIORef stref
   let (d,f) = processButton c state
   writeIORef stref (d,f)
   setValue display (show d) env
```

The operator `entryS` is similar to `entry` but has a further first argument to specify the initial layout of this widget (here: the background color). Note that we exploit the higher-order features of Curry to create the individual buttons by the generic function `cbutton` in a compact way. Each button has an associated event handler `buttonPressed` that reads the current state, modifies it, and shows the new operand in the entry widget referenced by the variable `display`. The actual update of the state depending on the selected button is computed by the operation `processButton`:

```
processButton :: Char -> (Int,Int->Int) -> (Int,Int->Int)
processButton b (d,f)
 | isDigit b = (10*d + ord b - ord '0', f)
 | b=='+'    = (0,((f d) +))
 | b=='-'    = (0,((f d) -))
 | b=='*'    = (0,((f d) *))
 | b=='/'    = (0,((f d) `div`))
 | b=='='    = (f d, id)
 | b=='C'    = (0, id)
```

Finally, the complete application is executed by evaluating the operation `main` that first creates a new `IORef` object and then runs the UI with this object:

```
main = do stref <- newIORef (0,id)
          runUI "Calculator" (calcUI stref)
```

We have already mentioned that the use of free variables as references to UI elements avoids the construction of wrong identifiers that might happen if strings are used as identifiers, as in scripting languages like Tcl/Tk, HTML/CGI, PHP, etc. Moreover, this also improves compositionality in the construction of UIs. For instance, if fixed strings are used as reference identifiers, there might be name clashes between different

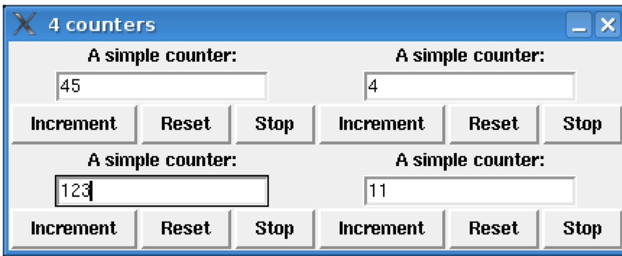

Fig. 3. A UI with four independent counters executed as a GUI

references when independent UIs are composed in a larger UI. Due to the use of free variables that represent fresh values every time they are introduced, such name clashes are avoided in our library. For instance, consider the simple counter UI above. Each use of `counterUI` introduces its own fresh local reference variable `val`. Thus, we can easily put four different counters in one UI by

```
counter4 = col [row [counterUI,counterUI],
                row [counterUI,counterUI]]
```

so that "`runUI "4 counters" counter4`" creates a UI with four independent counter UIs (see Fig. 3). This property of compositionality is particularly useful if one combines various UIs into complex web pages (see below).

The use of free variables for fresh references in data structures is a specific functional logic design pattern called "locally defined global identifier" [2]. An alternative would be a global counter to create unique references that is threaded through the construction of the user interface. Such an approach leads either to a monadic programming style with an imperative flavor [5,18] or puts some restrictions on the possible dependencies between input fields and buttons [25].

5 Implementation Issues

The definition of the components to specify a user interface, as discussed in Section 3, are contained in a library `UI` so that one has to import this library in order to define an interface. However, such an interface is not executable without specifying whether it should be run as a GUI or a WUI. For this purpose, our framework provides two implementations of the general UI concept by transforming UIs into GUIs or into WUIs. The necessary functionality is contained in the libraries `UI2GUI` and `UI2HTML`, respectively. In order to execute a UI as a GUI (as shown in the left-hand sides of Fig. 1 and 2), one has to import the library `UI2GUI` (which has the same interface as `UI`) instead of `UI`, i.e., one has to put the import declaration

```
import UI2GUI
```

at the beginning of the module containing the corresponding UI specification. In order to execute a UI as a WUI (as shown in the right-hand sides of Fig. 1 and 2), one has to replace `UI2GUI` by `UI2HTML` in the import declaration, and everything else is left

unchanged (apart from the command to generate an executable from the corresponding Curry program).

The implementation of the library `UI2GUI` is straightforward by exploiting the existing Curry library `GUI` [9] and mapping UI elements into corresponding GUI elements. Thus, the main function `runUI` is implemented in this library by transforming the main term and all its subterms of type `UIWidget` into the corresponding GUI widgets and then calling the main function `runGUI` of the `GUI` library.

The implementation of the library `UI2HTML` is more advanced since the existing Curry library `HTML` [10] does not support server interaction inside a web page. Since this is possible by the Ajax framework [7], we have added extensions to the `HTML` library (based on Ajax) to support the interaction model implied by the `UI` library. Based on these extensions, the main function `runUI` is implemented in the library `UI2HTML` by transforming terms of type `UIWidget` into corresponding HTML expressions that are put into an HTML form that contains the HTML input elements and JavaScript code to implement the interaction with the web server.

In typical web applications, a user interface is not the single entity of a web page but often embedded in a larger web page (containing headers, navigation bars, other input elements, explaining text, etc). In order to put UIs as elements into larger web pages, our library `UI2HTML` also exports a function `ui2hexps` that maps a UI specification into an HTML expression that can be inserted into an HTML page constructed with the `HTML` library [10]. Since the references used in UIs (of type `UIRef` discussed above) and the references used in the `HTML` library to access the values of the input elements are of different type[2], there are also conversion functions between these kinds of references. Thus, the values set in a UI can be used to influence values or elements in the surrounding web page, and vice versa.

6 Extended UI Programming

The structure of UI specifications is a generalization compared to previous proposals for GUI or WUI programming. In this section, we discuss two possibilities to extend previous more specialized approaches to interface programming by exploiting our UI approach.

6.1 Transforming GUIs into WUIs

Since the structure of UI elements is very similar to the elements of the Curry library `GUI`, which has been already used for various applications (e.g., [11,16,23]), one can also use our concept of UIs to enable the execution of such GUI-based desktop applications as web applications. For this purpose, we have also implemented a library `GUI2HTML` that provides the same interface as the library `GUI` but executes a GUI as a WUI by exploiting the library `UI2HTML`. For instance, we have used this implementation to execute the Curry analysis environment CurryBrowser (its implementation

[2] This is necessary because UI references must be more general in order to support their mapping into GUIs or WUIs.

consists of almost 4000 lines of Curry code), which is written in Curry and has a quite advanced graphical user interface (see [11]), in a standard web browser. The only necessary change was the replacement of the import of the library `GUI` by the import of the library `GUI2HTML` in the source code of the CurryBrowser implementation.

6.2 Type-Safe UIs

[12] presented a technique to construct type-safe WUIs in a high-level manner. The basic idea is to provide a set of typed WUIs for basic data types, like `wInt` for integers or `wString` for strings, and a set of combinators for typed WUIs, like `wPair` for pairs, `wTriple` for triples, `wList` for lists, etc. For instance, the expression "`wlist (wPair wInt wString)`" specifies a WUI to manipulate values of type `[(Int,String)]`. One of the important properties of such typed WUIs is the fact that the user can only enter values of the correct type, i.e., if the user attempts to enter ill-typed values, an error message appears and the user has to correct the value. Thus, the application program need not check the values, provide error messages etc. A further important aspect is the possibility to constrain the type of allowed values by any computable predicate. For instance, if the predicate `correctDate` checks whether a triple of integers forms a legal date, one can specify by

```
wTriple wInt wInt wInt `withCondition` correctDate
```

a WUI where one can enter only legal dates.

We can apply the same idea to UIs in order to obtain type-safe WUIs (similarly to [12]) as well as type-safe GUIs (which have not been considered before). Therefore, we have implemented two libraries `TypedUI2HTML` and `TypedUI2GUI` that provide almost the same interface as [12] (i.e., it has all the entities, like `wInt`, `wString`, `wPair`, for specifying typed UIs) and an operation `typedui2ui` to map a typed UI specification together with an initial (type-correct) value into a standard UI widget that allows only the manipulation of type-correct data. In addition, `typedui2ui` also returns operations to access, set, and update the value shown in the typed UI. For instance, the following program defines a UI containing a list (`xs`) of integers that can be together incremented or reset, and a button to compute their sum:

```
counters :: [Int] -> UIWidget
counters xs =
  col [label "A list of counters:", widget,
       row [button (updval (map (+1))) "Increment all",
            button (setval (repeat 0)) "Reset all",
            button compute "Compute sum:", entry sval ""]]
 where
  sval free
  (widget,getval,setval,updval) = typedui2ui (wList wInt) xs
  compute env = do cs <- getval env
                   setValue sval (maybe "" (show . sum) cs) env
```

Note that the derived operations `getval`, `setval`, and `updval` access or manipulate values of type `[Int]`, i.e., the implementation checks whether all widgets contain only

integer values (in contrast to the `counterUI` example in Section 4). As a consequence, `getval` returns a `Maybe` value, i.e., it returns `Nothing` if some of the current input fields contain illegal values. This is also the reason why the operation `compute` uses the standard function `maybe` in order to return the empty string as the sum value if the current content `cs` is `Nothing`. The result of this construction is a standard UI, i.e., we can create a type-safe GUI or WUI for a list of four integers by executing "`runUI "Counters" (counters [1..4])`".

7 Conclusions and Related Work

We described a framework to implement user interfaces in a high-level, declarative manner. Our approach is based on separating the structural, functional, and layout aspects of a user interface. We showed that the features available in functional logic languages can be exploited to provide appropriate specifications of these issues. The hierarchical structure of UIs can be easily specified as term structures. The associated functionality can be specified by attaching event handlers (i.e., functions) to the elements of these term structures. The connections of event handlers to the individual widgets of the UI can be described by logic variables. This avoids typical programming errors in untyped scripting languages and supports compositionality in the construction of complex UIs. Finally, the concrete layout is separated from the structural and functional aspects of the UI. This supports the use of the same UI specification in different contexts, i.e., one can create either graphical user interfaces for desktop applications or web-based user interfaces from such descriptions only by importing the appropriate libraries. This simplifies the programming efforts to combine the advantages of existing user interface technologies. Finally, our framework also enables the transformation of existing GUI applications into web applications, the embedding of UIs into arbitrary HTML pages, and the construction of type-safe UIs. Although this functionality is a distinctive feature of our approach based on declarative programming techniques, we discuss some related work in the following.

Approaches to construct UIs in a declarative manner have been intensively studied in the functional programming community, e.g., [5,18,20,24,25,26]. Although there are approaches to create GUIs for different platforms [18] from the same base code, none of them support the unified creation of GUIs and WUIs.

Adobe AIR[3] enables the use of the same base code to create applications that run in a web browser as well as on a desktop. In contrast to our approach, Adobe AIR is not based on standard features of web browsers but requires specific software to be installed on the client's side. Another related work is the Google Web Toolkit[4] (GWT). GWT is a framework to implement dynamic web pages for Java programs similarly to GUI programming in order to create highly interactive web applications with reasonable efforts. GWT does not support the use of the same program to generate both GUI and WUI applications in contrast to our approach where concrete implementations (GUIs or WUIs) are automatically inferred from a single UI description. Moreover, because of the applied declarative programming concepts, our concrete UI descriptions are more compact.

[3] `http://www.adobe.com/devnet/air/`

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

Another popular method to construct UIs are graphical editors that support the construction of the UI's layout, e.g., Cocoa's Interface Builder[5]. Similarly to our approach, such UI editors also advocate the separation of layout and functionality by binding the graphical UI objects to the code of the base application. Although these graphical editors are useful to define the layout of appealing UIs in a simple manner, the connection of a constructed UI with the application code is less trivial than in our event handler model using a single implementation language. Moreover, a textual representation of UIs as program entities is precise and compact (all information about the UI is contained in the program), and it allows the application of standard programming techniques to construct complex UIs from application-oriented UI elements, e.g., as shown in [12] or Section 6.2 above. Another possibility is the generation of the textual UI specification from the data model of the application, e.g., one could generate the UIs to manipulate the application data from an entity-relationship model, as in the Ruby on Rails framework[6] (a similar framework for Curry is currently being developed).

The various features of the declarative base language Curry, in particular, algebraic data types, functions as first class citizens, logic variables, and polymorphic types, have shown to be useful to support the high-level, compact, and reliable specification of UIs that can be used in different contexts. The implementation of our concept as sketched in Section 5 is freely available with the latest distribution of PAKCS [15]. For future work it might be interesting to explore whether the same or a slightly modified concept can be also used to create user interfaces for other architectures, e.g., mobile devices.

References

1. Antoy, S., Echahed, R., Hanus, M.: A Needed Narrowing Strategy. Journal of the ACM 47(4), 776–822 (2000)
2. Antoy, S., Hanus, M.: Functional Logic Design Patterns. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 67–87. Springer, Heidelberg (2002)
3. Braßel, B., Hanus, M., Müller, M.: High-Level Database Programming in Curry. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 316–332. Springer, Heidelberg (2008)
4. Cabeza, D., Hermenegildo, M.: Internet and WWW Programming using Computational Logic Systems. In: Workshop on Logic Programming and the Internet (1996), `http://clip.dia.fi.upm.es/Software/pillow/`
5. Claessen, K., Vullinghs, T., Meijer, E.: Structuring graphical paradigms in TkGofer. In: Proc. of the International Conference on Functional Programming (ICFP 1997), vol. 32(8), pp. 251–262. ACM SIGPLAN Notices (1997)
6. Fischer, S.: 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, New York (2005)
7. Garrett, J.J.: Ajax: A New Approach to Web Applications (2005), `http://AdaptivePath.com`
8. Hanus, M.: 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)

[5] `http://developer.apple.com/tools/interfacebuilder.html`
[6] `http://www.rubyonrails.org/`

9. Hanus, M.: A Functional Logic Programming Approach to Graphical User Interfaces. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 47–62. Springer, Heidelberg (2000)
10. Hanus, M.: High-level server side web scripting in curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
11. Hanus, M.: CurryBrowser: A Generic Analysis Environment for Curry Programs. In: Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE 2006), pp. 61–74 (2006)
12. Hanus, M.: Type-Oriented Construction of Web User Interfaces. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2006), pp. 27–38. ACM Press, New York (2006)
13. Hanus, M.: Multi-paradigm Declarative Languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
14. Hanus, M.: Putting Declarative Programming into the Web: Translating Curry to JavaScript. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007), pp. 155–166. ACM Press, New York (2007)
15. Hanus, M., Antoy, S., Braßel, B., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F.: PAKCS: The Portland Aachen Kiel Curry System (2008), `http://www.informatik.uni-kiel.de/~pakcs/`
16. Hanus, M., Koj, J.: An Integrated Development Environment for Declarative Multi-Paradigm Programming. In: Proc. of the International Workshop on Logic Programming Environments (WLPE 2001), pp. 1–14, Paphos, Cyprus (2001); Computing Research Repository (CoRR), `http://arXiv.org/abs/cs.PL/0111039`
17. Hanus, M. (ed.): Curry: An Integrated Functional Logic Language (Vers. 0.8.2) (2006), `http://www.curry-language.org`
18. Leijen, D.: wxHaskell – A portable and concise GUI library for Haskell. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, pp. 57–68. ACM Press, New York (2004)
19. López-Fraguas, F., Sánchez-Hernández, J.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
20. Meijer, E.: Server Side Web Scripting in Haskell. Journal of Functional Programming 10(1), 1–18 (2000)
21. Ousterhout, J.K.: Tcl and the Tk toolkit. Addison-Wesley, Reading (1994)
22. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, Cambridge (2003)
23. Sadeghi, P.H., Huch, F.: The Interactive Curry Observation Debugger iCODE. Electronic Notes in Theoretical Computer Science, vol. 177, pp. 107–122 (2007)
24. Sage, M.: FranTk - a declarative GUI language for Haskell. In: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 106–117. ACM Press, New York (2000)
25. Thiemann, P.: WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 192–208. Springer, Heidelberg (2002)
26. Vullinghs, T., Tuijnman, D., Schulte, W.: Lightweight GUIs for Functional Programming. In: Swierstra, S.D. (ed.) PLILP 1995. LNCS, vol. 982, pp. 341–356. Springer, Heidelberg (1995)
27. Wadler, P.: How to Declare an Imperative. ACM Computing Surveys 29(3), 240–263 (1997)