©Springer-Verlag
In *Proc. of the Second International Workshop on Practical Aspects of Declarative Languages, PADL'00*, Boston.
Springer LNCS 1753, pp. 47–62, 2000

A Functional Logic Programming Approach to Graphical User Interfaces

Michael Hanus*

Informatik II, RWTH Aachen, D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de

Abstract. We show how the features of modern integrated functional logic programming languages can be exploited to implement graphical user interfaces (GUIs) in a high-level declarative style. For this purpose, we have developed a GUI library in Curry, a multi-paradigm language amalgamating functional, logic, and concurrent programming principles. The functional features of Curry are exploited to define the graphical structure of an interface and to implement new graphical abstractions, and the logic features of Curry are used to specify the logical dependencies of an interface. Moreover, the concurrent and distributed features of Curry support the easy implementation of GUIs to distributed systems.

1 Introduction

The implementation of graphical user interfaces for application programs is a non-trivial task which is usually supported by specific libraries. Although it is clear that any serious programming language must have a library for implementing GUIs, there are many different approaches to structure those libraries. In this paper we propose a GUI library for integrated functional logic languages (see [6] for a survey) and show how the features of such integrated languages can be exploited to provide a nice structure for the implementation of GUIs.

In this paper, we consider the language Curry [7, 11], a modern multi-paradigm declarative language which integrates functional, logic, and concurrent programming paradigms. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). Curry also provides additional features in comparison to the pure paradigms (compared to functional programming: search, computing with partial information and constraints; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions, more flexible search strategies) and supports programming-in-the-large with specific features (types, modules, encapsulated search).

In order to avoid reinventing the wheel, our GUI library is based on Tcl/Tk [14]. The main purpose of this contribution is to provide a suitable structure to

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1 and by the DAAD under the PROCOPE programme.

```
runWidget "Hello"
      (TkCol [] [TkLabel [TkText "Hello world!"],
                 TkButton tkExit [TkText "Stop"]])
```

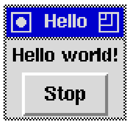


Fig. 1. A simple "Hello world" GUI

```
TkCol [] [
   TkEntry [TkRef val, TkText "0"],
   TkRow [] [TkButton (tkUpdate incrText val) [TkText "Increment"],
             TkButton (tkSetValue val "0")    [TkText "Reset"],
             TkButton tkExit                  [TkText "Stop"]]]
   where val free
```

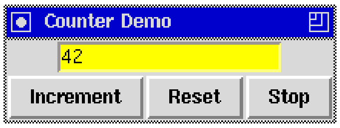


Fig. 2. A specification of a counter GUI

access the components of Tcl/Tk in a high-level way from Curry programs. We will see that the functional *and* logic features of Curry supports together a good structure to describe GUIs.

In order to get an impression of the proposed structure of GUI implementations, Fig. 1 shows a simple but complete implementation of a "Hello world" GUI based on our library. The GUI is started by the I/O action **runWidget** which takes a string (the title of the main window) and a specification of a GUI as an argument. This specification is basically a description of the hierarchical layout of the various GUI elements. In this simple example, the GUI is a column (**TkCol**) of two elements: a label element (**TkLabel**) containing a text and a button (**TkButton**) which terminates the GUI by the action **tkExit** when the button is pressed.

Beyond the hierarchical layout structure, GUIs have also a logical structure which connects the different elements of a GUI. For instance, different buttons refer to the manipulation of particular entry fields in a GUI. As a simple example, consider the counter GUI shown in Fig. 2. Since clicking the increment button should increase the value of the entry field by one, there is a connection between the action of the "Increment" button and the value shown in the entry field (and similar for the "Reset" button). Many GUI libraries (e.g., [14, 19]) solve this problem by forcing the programmer to assign explicit names (strings) to the different GUI elements which are subsequently used as references to them. Since these names are strings, often no consistency checks are done so that runtime errors can occur when a name is referred but does not exist. Moreover, new graphical abstractions which combine several elements are difficult to define since the necessary names can clash with other existing names. To avoid these problems, we use logical variables ("fixed but unknown widget references") to refer to the different GUI elements. If a reference to some GUI element is necessary, we introduce for this purpose a logical variable ("**TkRef val**" in Fig. 2) which can be used in actions like `tkSetValue` or `tkUpdate` to manipulate these elements. Thus, a GUI in our framework is a partially instantiated data structure[1] where multiple occurrences of a logical variable denotes the logical dependencies inside the GUI. In Fig. 2 the entry field showing the current value of the counter is referred by the logical variable `val`. Clicking the "Increment" button causes the invocation of the event handler "`tkUpdate incrText val`" that applies the function `incrText` (which increments the textual representation of a number by one) to the string shown in the entry element referred by `val`. Similarly, this field is set to the string `"0"` by pressing the "Reset" button.

2 Basic Elements of Curry

This section provides a brief overview of Curry as necessary to understand our approach to GUI programming. More details about Curry's computation model and a complete description of all language features can be found in [7, 11].

From a syntactic point of view, a Curry program is a functional program[2] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of functions and the data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with uninstantiated arguments (logical variables). The behavior of such function calls depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and functions for computing the concatenation of lists and the last element of a list:

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

conc :: [a] -> [a] -> [a]
conc eval flex

conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs   = x   where x,ys free
```

[1] Since the GUI library does not export any constructor for the argument type of `TkRef`, the type system ensures that no ground values can be inserted as arguments of `TkRef`.

[2] Curry has a Haskell-like syntax [15], i.e., (type) variables and function names 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 ("$f\ e$").

The data type declarations define `True` and `False` as the Boolean constants and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type `List a` is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[3] Since `conc` is explicitly defined as flexible[4] (by "`eval flex`"), the equation "`conc ys [x] =:= xs`" can be 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 "l `|` c `=` e `where` vs `free`" where l has the form $f\,t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, and vs is the list of *free variables* that occur in c and e but not in l (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 `Constraint`. Each Curry system must support 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 data terms (i.e., terms without defined function symbols). However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints for applications in operation research problems, as in the PACS implementation [9]. *Expressions* are of the following form:

$e ::=$	c	(constants like numbers or identifiers)
	x	(variables x)
	$(e_0\ e_1 \ldots e_n)$	(application)
	`if` b `then` e_1 `else` e_2	(conditional)
	e_1`=:=`e_2	(equational constraint)
	e_1 `&` e_2	(concurrent conjunction of constraints)
	e_1 `&>` e_2	(sequential conjunction of constraints)
	`let` $x_1, \ldots, x_n$ `free in` e	(existential quantification)

Curry has also a polymorphic type system which ensures that the expressions e, e_1, e_2 in the last three alternatives are always constraints.

The operational semantics of Curry, as precisely described in [7,11], is a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Due to the use of an optimal evaluation strategy [1], Curry can be considered as a

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

[4] As a default, all non-constraint functions are rigid.

generalization of concurrent constraint programming [17] with a lazy (optimal) evaluation strategy. Due to this generalization, Curry supports a clear separation between the sequential (functional) parts of a program, which are evaluated with an efficient and optimal evaluation strategy, and the concurrent parts, based on the concurrent evaluation of constraints, to coordinate concurrent program units.

Monadic I/O: Since the implementation of GUIs in a declarative language requires some knowledge about performing I/O in a declarative manner, we sketch the I/O concept of Curry which is identical to the monadic I/O concept of Haskell [20]. In the monadic approach to I/O, an interactive program computes a sequence of actions which are applied to the outside world. *Actions* have type "`IO` α" which means that they return a result of type α whenever they are applied to (and change) the outside world. For instance, `getChar` of type `IO Char` is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. Actions can only be sequentially composed. For instance, the action `getChar` can be composed with the action `putChar` (which has type `Char -> IO ()` and writes a character to the terminal) by the sequential composition operator `>>=` (which has type `IO` α `->` `(`α `-> IO` β`) -> IO` β), i.e., "`getChar >>= putChar`" is a composed action which prints the next character of the input stream on the screen. The second composition operator `>>` is like `>>=` but ignores the result of the first action. Furthermore, `done` is the "empty" action which does nothing (see [20] for more details).

Disjunctive computations: A difficulty in combining logic-oriented languages with I/O is the fact that the meaning of I/O operations becomes unclear when a computation is split into two disjunctive paths. In Curry this problem is solved by encapsulating possible non-deterministic computations between I/O operations (see [10] for details). We do not further discuss this technique here but remark that non-deterministic search is not performed for goals containing global variables but only for goals where all unbound variables are existentially quantified in this goal. Since we will create GUIs via global variables ("ports", see below), non-deterministic steps (i.e., a potential copying of GUIs in a disjunctive computation) are automatically avoided (i.e., suspended) if they include a reference to a GUI. This provides for a conceptually clean integration of GUI programming in a logic language (in contrast to low-level Tcl/Tk libraries like in Sicstus-Prolog).

3 Object-Oriented and Distributed Programming

GUI programming as proposed in this paper is based on the techniques for object-oriented and distributed programming in Curry. Therefore, we sketch these features in this section. More details and examples can be found in [8].

It is well known [18] that concurrent logic programming languages provide a simple way to implement (concurrent) objects. An object can be seen as a constraint or predicate processing a stream of incoming messages. The local state of the object is a parameter which may change in recursive calls when a message is processed. Thus, the general type of an object o is

```
o :: st -> [mt] -> Constraint
```

where *st* is the type of the local state and *mt* is the type of the messages which can be sent to the object. For instance, a simple counter object which understands the messages `Inc`, `Get` *v*, and `Stop` can be implemented in Curry as follows (the predefined type `Int` denotes the type of all integer values and `success` denotes the always satisfiable constraint):

```
data CounterMessage = Inc | Get Int | Stop

counter :: Int -> [CounterMessage] -> Constraint
counter eval rigid

counter n (Inc   : ms) = counter (n+1) ms
counter n (Get v : ms) = v=:=n & counter n ms
counter _ (Stop  : ms) = success
```

The type declaration for `counter` (which can be omitted since types are reconstructed in Curry by a type inference algorithm) specifies that a `counter` object keeps an integer as local state and understands messages of type `CounterMessage`. Since `counter` is declared as a rigid function, an expression "`counter` *n* *s*" can reduce only if *s* is a bound variable.

The evaluation of the constraint "`counter 0 s`" creates a new counter object with initial value `0` where messages are sent by constraining the variable `s` to hold the desired messages. For instance, the constraint

```
let s free in  counter 0 s  &  s=:=[Inc, Inc, Get x, Stop]
```

is successfully evaluated by binding `x` to the value 2.

In realistic applications, the stream of messages is not instantiated at once but incrementally constrained by various other objects (message senders). In order to allow a dynamic extension of senders and to ensure the sending of messages in constant time, Janson et al. [12] proposed the use of port constraints which have been generalized in Curry to provide a high-level approach to implement distributed systems [8]. In principle, a *port* can be considered as a multiset (of messages) where the individual elements are not directly accessible. There are two primitive constraints on ports, where "`Port a`" denotes the type of a port to which messages of type `a` can be sent:

```
openPort :: Port a -> [a] -> Constraint
send     :: a  ->  Port a -> Constraint
```

The evaluation of "`openPort p s`" where `p` and `s` are uninstantiated variables establishes a *port constraint* which is satisfied iff all elements in the port `p` also occur in the message stream `s` and vice versa. A message `m` is sent to the port `p` by evaluating the constraint "`send m p`" which constrains (in constant time) `p` and the corresponding stream `s` to hold the element `m`. From a logic programming point of view, `p` and `s` are partially instantiated variables that are more and more constrained by solving the constraint "`send m p`". In contrast to the purely functional part of Curry, the communication is performed in a strict manner to avoid a communication overhead in a distributed system, i.e., the message `m` is reduced to a data term before sending it.

With the use of ports, we can define a generic constraint `new`

```
new :: (st -> [mt] -> Constraint) -> st -> Port mt -> Constraint
new obj st p = let s free in  openPort p s &> obj st s
```

to create new objects with initial state `st` and communication port `p`. Thus,

```
let cp free in  new counter 0 cp  &  client1 cp  &  client2 cp
```

creates a counter with two different clients. Each client can increment the counter by solving the constraint "`send Inc cp`". The current state of the counter can be asked by "`send (Get x) cp`" so that `x` is unified with the current counter value. Thus, free variables in messages provide an elegant method to return values to the sender without explicitly creating reply channels.

In order to support the programming of distributed systems, where different components run on different machines in the Internet, ports can be declared as *external* so that they are accessible from outside. This feature together with concrete examples for distributed applications using ports can be found in [8].

4 A Functional Logic GUI Library

A main objective of our GUI library is a design which smoothly interacts with the features of the base language Curry. In particular, a careful design is necessary to deal with features like non-determinism and search. We solve this by using ports for GUI communication. Therefore, we introduce a new primitive I/O action

```
openWish :: String -> IO (Port TkMsg) .
```

"`openWish t`" creates a new GUI window with title `t` and returns a *communication port* for this GUI. The (abstract) data type `TkMsg`[5] is the type of possible messages for GUI communication. These are only used in the implementation of the GUI library but not visible to the user of the library. The important design issue is the fact that a GUI communication port is always external and created by such an I/O action. Since the GUI communication port is a global variable, disjunctive computations or search are not performed for subexpressions containing a reference to such a port (compare Sect. 2). This behavior is perfectly intended since it avoids the potential duplication of GUIs in different disjunctive branches of a computation. Nevertheless, the non-deterministic features of the base language can be used inside a GUI if the search computations are encapsulated and do not refer to the global port.

After the creation of a GUI communication port `gp`, we can run a GUI specification `gs` (like the one shown in Fig. 2) by the constraint "`runWidgetOnPort gs gp`". Basically, `runWidgetOnPort` communicates with the port `gp` by translating the GUI specification `gs` into appropriate Tcl commands (see Sect. 7). The I/O action `runWidget` (see Fig. 1) composes the functionality of `openWish` and `runWidgetOnPort`: it creates a new GUI communication port and runs the GUI specification on this port. Note that `runWidget`

[5] Most of the identifiers defined in the GUI library are prefixed by `Tk` since the library is based on the Tcl/Tk toolkit. Similarly, the name `openWish` refers to the fact that the windowing shell `wish` is used for the communication with the Tk toolkit.

executes a GUI as an I/O action whereas `runWidgetOnPort` executes a GUI as a (concurrent) constraint. Therefore, `runWidget` is usually applied when one GUI is executed as the main program (Fig. 1), whereas `runWidgetOnPort` is applied when one GUI should be executed concurrently to other activities (e.g., other concurrent objects or GUIs, see Sect. 5).

Layout structure of a GUI: A GUI specification is a description of the hierarchical layout structure of the GUI together with the actions that should be performed when, for instance, a GUI button is pressed. To be more precise, a GUI specification is a term of the following data type (here we list only the widgets used in the examples of this paper):

```
data TkWidget a = TkButton (Port TkMsg -> a) [TkConfItem a]
                | TkCheckButton              [TkConfItem a]
                | TkEntry                    [TkConfItem a]
                | TkLabel                    [TkConfItem a]
                ⋮
                | TkRow [TkCollectionOption] [TkWidget a]
                | TkCol [TkCollectionOption] [TkWidget a]
```

Thus, a GUI specification is a simple widget (like a button or entry), a row (`TkRow`) or a column (`TkCol`) of widgets.[6] The first parameter of `TkRow`/`TkCol` specifies additional options for the geometric alignment for widget composition, like centering, left alignment, expanding subwidgets if extra space is available:

```
data TkCollectionOption = TkCenter | TkLeft | ... | TkExpand
```

A button widget (`TkButton`) is intended to perform an action whenever the user presses this button. Therefore, an event handler is associated to each button widget (first parameter). Other widgets can also contain event handlers but they are optionally associated in the list of configuration items (see below). Since these event handlers are responsible for an event of a specific GUI, *event handlers* have type "`Port TkMsg -> a`" where `a` is the result type of the event handler which is either `Constraint` (for GUIs executed concurrently to other objects) or `IO ()` (for GUIs executed as an I/O action). Consequently, this type variable is also a parameter for the entire GUI structure.

Logical structure of a GUI: Before discussing event handlers in more detail, we must understand the concept to describe the logical structure of GUIs. As mentioned in the introduction, GUIs have a layout structure *and* a logical structure. While the layout structure is simply described by composing simple widgets into widget collections (`TkRow` and `TkCol`), the logical structure contains dependencies between different widgets and their event handlers. For instance, pressing some button usually results (after some computation) in the update of one or more other widgets. Although many GUI libraries (e.g., [14, 19]) are based on user-selected strings to identify the different widgets, we propose to use logical variables to refer to individual widgets which avoids many programming errors

[6] The row/column organization of widgets is sufficient for our purposes but one can also extend the library to cover other forms of widget collections (see also Sect. 6).

and provides for better abstractions. For this purpose, each primitive widget can have a number of items to configure the widget, like[7]

```
data TkConfItem a =
         TkRef TkRefType          -- a reference to this widget
       | TkText String            -- an initial text contents
       | TkWidth Int              -- the width of a widget
       | TkBackground String      -- the background color
       | TkCmd (Port tkMsg -> a)  -- an associated event handler
       ⋮
```

Most of these configuration items directly correspond to similar options in the Tk toolkit with the exception of `TkRef`. Since `TkRefType`, the type of all widget references, is abstract, i.e., no constructors of this data type are available to the user of the GUI library, the only reasonable way to use the `TkRef` item is with a free logical variable as shown in Fig. 2. If we run a GUI specification on a concrete port, this variable will be instantiated to a unique widget reference which is not visible to the user. The important point is that this variable can also be used in event handlers for other widgets in the same GUI. For this purpose, there are the following primitives to construct event handlers for GUIs:

```
tkExit    :: Port TkMsg -> IO ()
tkGetValue:: TkRefType -> Port TkMsg -> IO String
tkSetValue:: TkRefType -> String -> Port TkMsg -> IO ()
tkUpdate  :: (String->String) -> TkRefType -> Port TkMsg -> IO ()
```

`tkExit` terminates the GUI, `tkGetValue` gets the (String) value currently stored in the widget referred by its first argument, `tkSetValue` sets the value stored in the referred widget, and `tkUpdateValue` updates the value according to an update function. The same set of primitives is also available for GUIs executed as a concurrent constraint:

```
tkCExit     :: Port TkMsg -> Constraint
tkCGetValue :: TkRefType -> Port TkMsg -> String -> Constraint
...
```

The event handlers attached to some widget are automatically invoked with the current GUI communication port whenever a GUI specification is executed by `runWidgetOnPort` (or `runWidget`), see also the examples in Fig. 1 and 2. Thus, a GUI specification is executed by sending commands that create the widget layout through the communication port followed by a scheduler which invokes the corresponding event handlers whenever the user performs some action on the GUI (see Sect. 7 for more details).

To change the configuration of widgets dynamically (e.g., changing colors, deactivating or activating buttons and entries), there is also a primitive

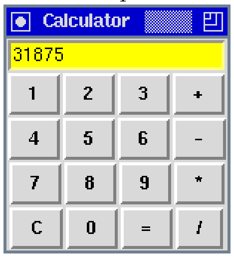
```
tkConfig :: TkRefType -> TkConfItem a -> Port TkMsg -> IO ()
```

which adds a configuration item to a particular widget in a GUI.

[7] Note that not all configuration items are meaningful for all widgets. This is checked at run time in our library, but in can be also checked at compile time with a more sophisticated type system, as proposed in [3].

5 Example: A Calculator

We have already seen in Fig. 1 and 2 two specifications of simple GUIs using our library. However, many interactive applications contain a state which is shown and modified by a GUI. To demonstrate the implementation of these kinds of applications with our GUI concept, we present in the following the implementation of a simple calculator GUI as shown to the right. We model the calculator as an object which accepts the following messages:




```
data CalcMsg = Button Char | Display String
```

The message "`Button c`" is sent whenever the button `c` (e.g., `'1'`, `'2'`,...,`'+'`, `'*'`,...) is pressed. The message "`Display s`" is sent to get the current value of the operand, i.e., the argument `s` (which is usually an unbound variable) is instantiated with the current operand of the calculator. The calculator's local state is a pair `(d,f)` with the current operand `d` and an accumulator function `f` to be applied to `d` (this idea is due to [19]). With the techniques sketched in Sect. 3, we can implement calculator objects as follows (the rigid Boolean function `==` tests the equality of two ground expressions, i.e., e_1`==`e_2 reduces to `True` if both e_1 and e_2 are reducible to identical ground data terms; (e op) denotes the partial application of the operator op to the left argument e):

```
calcMgr :: (Int,Int->Int) -> [CalcMsg] -> Constraint
calcMgr eval rigid
calcMgr (d,f) (Display s : ms) = s=:=(show d) &> calcMgr (d,f) ms
calcMgr (d,f) (Button b  : ms)
 | isDigit b = calcMgr (10*d + ord b - ord '0', f) ms
 | b=='+'    = calcMgr (0, ((f d) +)) ms
 | b=='-'    = calcMgr (0, ((f d) -)) ms
 | b=='*'    = calcMgr (0, ((f d) *)) ms
 | b=='/'    = calcMgr (0, ((f d) `div`)) ms
 | b=='='    = calcMgr (f d, id) ms
 | b=='C'    = calcMgr (0, id) ms
```

Since the GUI needs a reference to the calculator object, we add it as a parameter `cm` to the GUI:

```
calc_GUI cm = TkCol [] [TkEntry [TkRef display, TkText "0"],
                        TkRow [] (map cbutton ['1','2','3','+']),
                        TkRow [] (map cbutton ['4','5','6','-']),
                        TkRow [] (map cbutton ['7','8','9','*']),
                        TkRow [] (map cbutton ['C','0','=','/'])]
  where display free
        cbutton c = TkButton (button_pressed c) [TkText [c]]
        button_pressed c gp = let d free in
                                 send (Button c) cm &>
                                 send (Display d) cm &>
                                 tkCSetValue display d gp
```

Here we exploit the higher-order features of the base language: To create the individual buttons, we use a generic function `cbutton` which is mapped on the particular lists of characters. The event handler `button_pressed` for each button sends a corresponding `Button` message to the calculator and shows the new operand of the calculator in the `display` widget. A new calculator application on a given GUI communication port `gp` is created by the following function:

```
runCalcOnPort gp
 | let cm free in
   new calcMgr (0,id) cm & runWidgetOnPort (calc_GUI cm) gp
 = done
```

Now the complete application is started by

```
openWish "Calculator" >>= runCalcOnPort
```

This implementation is modular similarly to the classical model-view-controller paradigm of Smalltalk-80 [13]. The application (represented by `calcMgr`) is completely independent to the user interface. All the programming techniques of the base language (laziness, higher-order functions, constraints, search etc.) can be used to implement the application. Due to the independence of the user interface and the application, it is also possible to have several GUIs (which represents the applications in different ways) for one application. In our implementation above, this is easily possible by changing the function `runCalcOnPort` to start one application together with several concurrent GUIs. This feature of our GUI design is also useful for developing GUIs for distributed applications where the GUI shows and manipulates different components of a distributed system. For instance, we have implemented a GUI for sending emails where the email address can be inserted by querying an address server running on some other machine. Due to lack of space, we omit a concrete example for this, but from the previous example it should be obvious how to use the distributed features of Curry (see Sect. 3 and [8]) in GUIs.

6 Application-Oriented Extensions

This section shows how the features of the base language can be exploited to define new application-oriented graphical elements for GUIs. As a simple example (which is often predefined in GUI libraries), consider the implementation of a radio button column as a new GUI element. A radio button column is a column of check buttons where at most one button is "on", i.e., if the user activates a button in this column, all other buttons must be set to "off" (for the sake of simplicity, the values "off" and "on" are represented by the strings `"0"` and `"1"`). This can be implemented by the following function, where "`radioButtonCol r labs cmd`" creates a new radio button column with reference `r`, labels `labs` (i.e., the strings shown at each button) and event handler `cmd` which is called whenever the user presses a button (the auxiliary functions `gen_vars` n returns a list of n unbound variables, l`!!`i returns the i-th element of the list l, and `remove` i l removes the i-th element from the list l):

```
radioButtonCol r labs cmd
 | r =:= gen_vars (length labs)  = TkCol [TkLeft] (gen_rb 0)
 where gen_rb i = if i==(length labs) then []
        else TkCheckButton [TkText (labs!!i), TkRef (r!!i),
                            TkCmd (rbcmd (r!!i) (remove i r) cmd)]
             : gen_rb (i+1)

rbcmd sel oth cmd gp =
 tkGetValue sel gp >>= \sv ->  --get state sv of this checkbutton
 (if sv=="1" then foldr (>>) done (map (\o->tkSetValue o "0" gp) oth)
             else done ) >>
 cmd gp
```

Thus, each button of a radio button column is a check button with an event handler which sets the other buttons (`oth`) to "off" whenever it is turned on, followed by the execution of the event handler `cmd` for the radio button. Two operations are important on radio buttons: get the index of the activated button in the column (or `-1` if there is no active button) and activate a particular button. These operations can be defined as follows:

```
getRadioValue [] _ = return (-1)
getRadioValue (r:rs) gp = tkGetValue r gp >>= \rval ->
  if rval=="1" then return 0
               else getRadioValue rs gp >>= \rspos ->
                    return (if rspos>=0 then rspos+1 else -1)

setRadioValue [] _ _ = done
setRadioValue (r:rs) i gp =
  tkSetValue r (if i==0 then "1" else "0") gp >>
  setRadioValue rs (i-1) gp
```

Due to the functional dimension of the base language, we can use radio button columns like any other widget in GUI specifications. For instance, a "traffic light GUI" as shown to the right, where the user can click on two traffic lights and the program ensures the pairwise exclusion of both red and green lights, is implemented by the following simple GUI specification:


```
TkRow [] [radioButtonCol l1 ["Red","Yellow","Green"] (ex l1 l2),
          radioButtonCol l2 ["Red","Yellow","Green"] (ex l2 l1)]
  where l1,l2 free

ex l1 l2 gp = getRadioValue l1 gp >>= \sel ->
             if sel>=0 then setRadioValue l2 (2-sel) gp else done
```

The event handler `ex` ensures that, whenever the user selects `Red` (`Yellow`, `Green`) for one traffic light, the other light switches to `Green` (`Yellow`, `Red`).

In a similar way, one can implement other more advanced graphical abstractions. For instance, one can define sets of radio buttons which are not simply mutually exclusive, like in the traffic light example, but must satisfy more complex constraints ("constraint buttons"). Due to the constraint logic programming features of Curry, such abstractions are fairly easy to implement. The usefulness of constraints in the design of user interfaces has been discussed elsewhere [16].

Note the importance of the use of free logical variables for widget references to built new graphical abstractions. If one assigns fixed strings to refer to widgets, as for instance in [14, 19], name conflicts can easily occur.

7 Implementation

The entire GUI library is implemented in Curry based on the connection to the Tcl/Tk toolkit [14]. The only extension which has been added to Curry is the connection to a windowing shell `wish` via the I/O action `openWish` (see Sect. 4). The messages sent to this port are of the following type:

```
data TkMsg = TkPut String | TkGet String
```

If the message "`TkPut s`" is sent to the port, the string `s`, which must be a valid Tcl command, is added to the input stream of the `wish`. On the other hand, the message "`TkGet s`" unifies the argument variable `s` with the next line of the output stream of the `wish`. These two messages are sufficient to implement `runWidgetOnPort` in Curry. First of all, the GUI specification is translated into a Tcl/Tk script to create the GUI layout which is sent to the `wish` via the message `TkPut`. Additionally, `runWidgetOnPort` creates a call-back list for handling the GUI events. If the user manipulates the GUI (e.g., press a button) so that an event occurs for which an event handler is defined, the `wish` emits a message on its output stream. This message is analyzed by the scheduler in `runWidgetOnPort` which calls the responsible event handler stored in the call-back list. Furthermore, the primitive event handlers like `tkGetValue` and `tkSetValue` are implemented by sending Tcl/Tk scripts that set or extract the corresponding GUI values.

Although all communication between the Curry system and Tcl/Tk is done by strings which have to be interpreted on both sides, this kind of implementation is efficient enough in our practical experiences. This is due to the fact that the communication in interactive applications tends to be slow since the user is usually much slower than the system. Therefore, the connection to Tcl/Tk through a port is sufficient for implementing practical systems. Furthermore, this technique supports an easy reuse of this library in other Curry implementations.

8 Conclusions and Related Work

We have presented a library for implementing GUIs in the functional logic language Curry. We have exploited the functional as well as the logic features of Curry for the design of the library. The functional features are used to define the layout structure of GUIs and to built new application-oriented graphical abstractions. The logic features (logical variables) are used to specify the logic dependencies inside a GUI. This allows a compact and readable specification of GUIs as expressions of a particular data type rather than a sequence of actions to built the GUI. As far as we know, this is the first approach to design a functional logic GUI library. Nevertheless, we want to relate it to some other approaches for GUI programming in declarative languages.

TkGofer [3] extends Gofer, a lazy functional language similar to Haskell, with a library for GUI programming. Similarly to our approach, the implementation is based on Tcl/Tk. TkGofer uses a monadic approach for GUI programming which forces the user to an imperative style. The different widgets are defined in a flat and sequential order and are later composed to a hierarchical layout structure by `pack` operations, similarly to Tcl/Tk scripts. This has the disadvantage that the layout structure is not defined together with the individual widgets and each widget must be given a name—even if they are used only once like the labels or buttons in Fig. 1 and 2. Furthermore, TkGofer is based on a sequential functional language. Thus, logic programming techniques like constraint solving as well as features for concurrent and distributed programming are not available.

The same holds for Fudgets [2], a GUI concept for lazy functional languages. Fudgets are processes accepting messages (for manipulating the state) and delivering messages (for sending information about an event). Primitive fudgets, like buttons, text inputs etc., are composed to more complex entities by connecting the input and output streams of the different fudgets. This approach makes it necessary to pass the GUI events via streams to the corresponding widgets to be manipulated by these events, which leads to less intuitive GUI specifications than using direct references to the corresponding widgets as in our approach.

Haggis [5] is a further GUI framework for Haskell. It is based on monadic I/O and uses concurrent processes to handle events. Instead of specifying a handler to be invoked when an event occurs, a new process is created that waits for this event. This has the drawback that widgets have two handlers in Haggis (one for the layout and one for the event handling).

Our proposal for GUI programming is not just an adaptation of existing GUI library designs to an integrated functional logic language, but it exploits the features of such an integrated language to support simple and readable GUI specifications. Moreover, the features of the base language like higher-order functions, constraints, and concurrency can be used to build new application-oriented graphical abstractions in a simple way and to support a modular connection of the application program to the user interface. In particular, the distribution features of Curry largely simplify the implementation of user interfaces for distributed systems.

The current definition of Curry has also some limitations which restricts the design of our GUI library. Currently, Curry has a Hindley-Milner like polymorphic type system [4]. It has been shown in [3] that a richer type system including type classes can improve the structure of a GUI library so that more errors can be caught at compile time. Since this is independent on the design issues discussed in this paper, we plan for the future to refine the design of our GUI library with a more sophisticated type system. Another topic for future work is to add the possibility to dynamically change the layout structure of GUIs which is currently not supported.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
2. M. Carlsson and T. Hallgren. Fudgets - A Graphical User Interface in a Lazy Functional Language. In *Conference on Functional Programming and Computer Architecture (FPCA'93)*. ACM Press, 1993.
3. K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proc. of the International Conference on Functional Programming (ICFP'97)*, pp. 251–262. ACM SIGPLAN Notices Vol. 32, No. 8, 1997.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Symposium on Principles of Programming Languages*, pp. 207–212, 1982.
5. S. Finne and S. Peyton Jones. Composing Haggis. In *Proc. of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*. Springer, 1995.
6. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
7. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pp. 80–93, 1997.
8. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
9. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PACS: The Portland Aachen Curry System. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/pacs/`, 1999.
10. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
11. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.5). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.
12. S. Janson, J. Montelius, and S. Haridi. Ports for Objects in Concurrent Logic Programs. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
13. 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.
14. J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, 1994.
15. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
16. M. Renschler. Configuration Spreadsheet for Interactive Constraint Problem Solving. In *Proc. of the ComputlogNet Industrial Conference on Advanced Software Applications*, Manchester, 1998.
17. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
18. E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pp. 251–273. MIT Press, 1987.
19. T. Vullinghs, D. Tuijnman, and W. Schulte. Lightweight GUIs for Functional Programming. In *Proc. 7th Int. Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pp. 341–356. Springer LNCS 982, 1995.
20. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.