

Lightweight GUIs for Functional Programming

Ton Vullingsh, Daniel Tuijnman, Wolfram Schulte

Fakultät für Informatik, Universität Ulm, D-89069 Ulm, Germany
email: {ton,daniel,wolfram}@informatik.uni-ulm.de

Abstract. Graphical user interfaces (GUIs) are hard to combine with functional programming. Using a suitable combination of monads, we are able to tame the imperative aspects of graphical I/O in a straightforward and elegant way. We present a concept to integrate lightweight GUIs into the functional framework, together with a library of basic functions and layout combinators to manipulate the GUI. An implementation of this library, using a set of high-level graphical I/O routines, is outlined. Examples demonstrate the simple way in which applications can be written.

1 Introduction

Everybody wants to use graphical user interfaces. And everybody wants to use functional programming languages. Unfortunately, these concepts are hard to combine: I/O, and graphical I/O in particular, is imperative in nature, and thus contradictory to the functional paradigm. Our goal is to reconcile these two concepts, by developing a framework in which a functional programmer smoothly can provide his program with a graphical user interface.

There are various ways to integrate I/O in general with a functional language [NR94, PJW93]. We have chosen for monadic I/O. This choice is firstly motivated by the advantage that monadic I/O can be combined with threading a global state through programs. Secondly, monadic programming expresses the sequentialization of I/O actions better than other styles do. This makes monads the natural choice for our system.

For the implementation of graphical I/O we have deliberately chosen not to reinvent the wheel, but to use an existing high quality tool, Tcl/Tk [Ous94], instead. This tool visualizes the user interface and handles events from the console and it communicates with the functional program in the form of strings. Thus we have reduced, on the side of the functional program, graphical I/O to textual I/O — for which I/O primitives are provided in, e.g., Glasgow Haskell [HPJW92] and Gofer [Jon93b].

We define a library of functions for manipulating the GUI. The programmer can write the GUI for his application in a structured and well-defined way, without having knowledge of the tool used. Also, the communication that takes place between the tool and the functional program is completely hidden for the user.

Advantages of our approach:

- *Easy to use.* Thanks to the convenient abstraction level of the library functions the main problems one has to consider upon creation of the GUI are aesthetical ones.
- *Unity of framework.* The application programmer can now write the user interface and the application proper in the same language.
- *Modularity.* The GUI state and the application state are independent. This makes it possible to write modular, reusable building blocks to create larger applications.
- *Generally applicable.* As long as monadic I/O is supported, our framework can be implemented in any functional language, lazy or eager.
- *Simple to implement.* Except for a few lines, we have written all the code in Gofer. Moreover, it is easy to extend the libraries.

Organization of this paper. Section 2 describes the way monads are used to reach the desired abstraction level. Section 3 gives a glance at the structure and interface of the functional GUI library. Then we present example applications in Sect. 4. Relevant implementation aspects are discussed in Sect. 5. A comparison with related work is given in Sect. 6, and finally, conclusions are drawn in Sect. 7.

Notation. The programs in this paper are written in Gofer [Jon93b], which is a subset of Haskell [HPJW92]. Characteristics of the syntax are: lambda abstraction $\lambda x \rightarrow$, the use of function names as infix operators by placing them between backquotes, and the use of placeholders $_$ for values that are discarded.

2 Monads

Since the seminal work of Moggi [Mog89], monads have become increasingly popular. After Wadler's paper [Wad90] on their use in functional programming, numerous applications have been published, e.g., [Wad92, JD93, KL94, LPJ94, Wad94].

In general, a monad serves to record *side-effects* of functions. A good example is the *I/O monad* as it is implemented in the Glasgow Haskell compiler [PJW93] — and recently laid down in the new definition of the Haskell language — and in the Gofer language [Jon93b]. Another one is the *State Reader monad* that enables the user to thread the reading of a state through his program. Likewise, there are monads to handle exceptions, non-determinism, continuations, etc.; but only the I/O and State Reader monads are relevant for the sequel.

2.1 Concepts and Laws

A monad consists of three parts: a parameterized datatype M and two polymorphic functions

```
result :: a -> M a
bind   :: M a -> (a -> M b) -> M b
```

The datatype `M a` encapsulates a value of type `a` into some action, e.g., an I/O operation. Using `bind`, one performs the action, retrieves the proper value and passes that to a function into the monad. The `result` function encapsulates a value into a void action.

When programming with monads, one writes functions with typing `a -> M b` and strings them together with `bind`. So, `bind` replaces function application which is reflected by the *monad laws*: `bind` must be associative, and `result` is its left and right identity.

In this paper, we use the `do` notation as described by [Jon93a]. An expression `do {e1; x <- e2; e3}` replaces the equivalent `e1 `bind` _ -> e2 `bind` \x -> e3`, i.e., the subexpression `x <- e2` binds `x` to the result of `e2`, whereas the result of subexpression `e1` is discarded.

2.2 Monadic I/O and Mutable Variables

Clearly, we need to perform I/O. To this end, we use the predefined `IO` monad, which comes with the primitive functions

```
putchar :: Char -> IO ()
getch   :: IO Char
```

for writing and reading of characters.

Secondly, we need some means of storing a state. Therefore, we use the *mutable variables* [LPJ94] as defined in Glasgow Haskell and Gofer, which also act on the `IO` monad.¹ A mutable variable is manipulated with the functions

```
newVar   :: a -> IO (Var a)
readVar  :: Var a -> IO a
writeVar :: Var a -> a -> IO ()
```

for creating one with an initial value, for reading its value, resp. for writing a new value to it. Of course, when multiple functions want to access a state stored in a mutable variable, the variable must be passed to these functions as an explicit argument.

As an example, consider a program fragment for a simple adder. The state consists of an integer, viz. the running total. The function that reads a number and calculates the new running total then looks like:

```
add :: Var Int -> IO ()
add v = do { tot <- readVar v ; s <- getLine
            ; newtot <- result (tot + numval s)
            ; putLine (newtot) ; writeVar v newtot
            }
```

First, the value of the running total is fetched from the variable `v` and the next line is read. Then the new running total is calculated; it is written on the console, and stored in the same variable `v`.

¹ Actually, they are defined more generally, but for our purposes this simplified version suffices.

2.3 Combining Monads: Monadic I/O and State Reader

Next, we integrate the GUI into the monadic framework. As the GUI will be changed dynamically, part of it must be stored — in a mutable variable. This variable must be accessible anywhere in the program, hence it must be threaded through all functions that are provided to the application. This problem is not specific for GUIs: any library that gives access to some global state, e.g. a library of database functions, must cope with this.

Therefore, we need the state reader monad which consults some fixed environment. As our environment is a variable, it is fixed, though its contents may change. Hence, to consult it, we do not need a state transformer, but a state reader will suffice. As we now have two monads, IO and State Reader, we need to combine them. This can be done in the following way [BW85, JD93]:

```
data RIO s a = RIO (s -> IO a)

bindRIO :: RIO s a -> (a -> RIO s b) -> RIO s b
bindRIO (RIO x) f = RIO (\s -> x s `bindIO` \a ->
                          let RIO g = f a in g s)

embIO2RIO :: IO a -> RIO s a
embIO2RIO a = RIO (\s -> a)
```

The definition of `bind` for `RIO` takes care that the same state `s` is used by `x` and again by `f`, and that the corresponding I/O operations are performed sequentially. The described functions that act on the IO monad, such as `putchar` and `newVar` can be embedded into the `RIO` monad with the function `embIO2RIO`. The embedded counterparts of `newVar`, `readVar` and `writeVar` will appear in the sequel as `newState`, etc. The programmer needs these functions to manipulate his own (application) state which also must be stored in a `Var`.

Thus, the GUI state remains completely hidden for the programmer; we only provide him with the datatypes

```
type GIO a = RIO (Var GUI) a
type Action = GIO ()
```

and the corresponding monad functions on `GIO`. The type `Action` is useful as most actions on the GUI only have a side effect and no proper result. The programmer only has to worry about defining his own state(s), global to the whole application or local to a window, as illustrated in Sect. 4.

3 A Functional Library for Graphical I/O

This section presents a relevant subset of our Gofer library for graphical user interfaces. A GUI consists of entities like windows and buttons, on which events are defined. These events are caused by the user or some other external process (e.g. a clock) and result in some action. The underlying application that is controlled in this way is called event driven.

Writing a library for graphical I/O is a complex and intensive job to do. In our approach we therefore decided to use an existing library of graphical I/O routines and to integrate this library within our concept. Because of its elegance and power we based our implementation on Tcl/Tk [Ous94] (see Sect. 5). Our library offers a high level interface to this tool by providing a powerful set of widget constructors and layout combinators.

3.1 Windows and Widgets

The basic building blocks of a graphical user interface are windows and widgets. A window is a container for widgets. A widget is a graphical entity with a particular appearance and behaviour. We can distinguish several kinds of widgets like buttons, labels, and entries.

Windows. A user interface may contain one or more windows. Possible actions on windows are closing and opening.

```
openWindow  :: [Config] -> Widget -> Action
closeWindow :: Ident  -> Action
```

```
type Ident = String
```

These functions yield actions. An action essentially produces side effects, i.e., changes to the GUI (cf. Sect. 2.3).

Windows are identified by a unique name. This name can be specified in the configuration list (see below). When closing a window, this name must be provided.

Widgets. In our view, a widget is either atomic or composite. For composite widgets, see Sect. 3.2. Widgets form an abstract datatype. For each atomic widget type, our library offers a constructor function:

```
buttonW, labelW, entryW, textW :: [Config] -> Widget
menuW  :: [Config] -> [MenuItem] -> Widget
```

In this paper, we use buttons, labels (static text of one line), entry fields (edit text of one line), text fields (edit text of several lines) and pull-down menus.

Specifying the external outline of individual widgets and windows is done by giving appropriate values for the configuration options: the name identifying the window or widget, the textual contents, the command to be performed upon pressing a button, etc.:

```
data Config = Name Ident | Text String | Command Action | ...
```

If the widget is identified with a name, the configuration of the widget can be read or modified, e.g., to read or write its textual contents, the functions

```
getText :: Ident -> GIO String
setText :: Ident -> String -> Action
```

are given.

3.2 Layout Combinators

Widgets can be composed vertically and horizontally using layout combinators and functions. Our basic combinators are

`(<<)`, `(^^)` :: `Widget -> Widget -> Widget`

These combinators are associative and have the following meaning:

- `v << w` places widget `w` to the right of widget `v`;
- `v ^^ w` places widget `w` below widget `v`.

The resulting new widget is called the father of `v` and `w`.

With every widget we can associate an inherited and an occupied area. The inherited area is the area a widget gets from its father. The occupied area is actually used for displaying information, and is always a centered subarea of the inherited one.

Initially, the occupied and inherited area equal the minimum dimensions needed by the widget to display its information. After combination with some other widget, the occupied area of the father is minimal again. His concatenated sons are placed in the left uppermost corner of his occupied area. If widget `v` is bigger than widget `w`, the inherited area of `v` will equal its occupied area, and the inherited area of `w` will equal the rest of the occupied area of the father.

The fill functions make a widget occupy its inherited area either horizontally (`fillX`) or vertically (`fillY`). The `expand` function makes a widget claim from its father all occupied area that is not inherited by one of his (other) sons.

`fillX`, `fillY`, `fillXY`, `expand` :: `Widget -> Widget`

In Fig. 1 we see three possible layout situations after application of (variants of) the combinators and fill functions. In the first picture, `A` and `B` are composed horizontally. Together they are combined vertically with `C`. In the second one, we let `A << B` and `C` occupy their inherited area in a horizontal direction. As a result of this, the father of `A` and `B` grows over the full length of `C`. In the third one, we let `A` and `B` grow vertically.

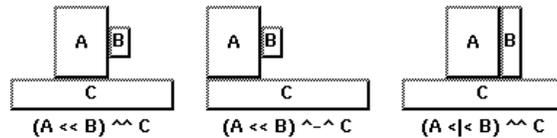


Fig. 1. Layout combinators and fill functions

In Fig. 2 we show the result of expanding widgets. In the first picture, we let `A` claim and take the area of its father. Likewise, in the second one, this area is

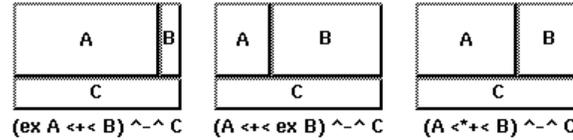


Fig. 2. Layout combinators and expand functions

claimed and taken by B. Finally, in the last picture, the area is claimed, taken and divided by both A and B.

In these examples, `ex` abbreviates `expand`, and the additional combinators are defined by:

```

( $\wedge\wedge$ ) = both ( $\wedge\wedge$ ) fillX      (<<+) = both (<<) fillXY
(<|<) = both (<<) fillY      (<*\<) = both (<<) (expand . fillXY)

```

```
both comb f a b = f a `comb` f b
```

Using the above sketched layout mechanism and combinators we can lay out user interfaces in an almost arbitrary way. Additionally, we can develop more powerful layout functions:

```

hfill :: widget
hfill = (expand.fillX) (labelW [])

matrix :: Int -> [Widget] -> Widget
matrix n = foldl1 ( $\wedge\wedge$ ) . map (foldl1 (<<)) . splitSegs n

```

The function `hfill` stretches an ‘empty’ widget as much as it can, pushing everything else aside. The function `matrix` takes a number of columns and a list of widgets and composes them in row major order.

The above gives us a basic library with which the user can already write his window-oriented applications. It may be extended with more widget types or configuration options, etc. But the essential structure remains the same.

3.3 Running the GUI

To illustrate the above defined functions, let us consider a minimal “hello world” application, see Fig. 3. If the user presses the button Hello Gofer!, the window is closed and the application terminates.

In Gofer, a program must be called `main` and have type `I0 ()`. In our concept, `main` must call the library function `startProg` with an initial window setup. It sets up the user interface and starts the event loop. The first window that is opened is called the main window. If the main window is closed, termination follows.

```

main :: IO ()
main = startProg w where
  w = openWindow [Name "h", Title "Hello"]
      (buttonW [Text "Hello Gofer!",
                Command (closeWindow "h")])

```



Fig. 3. *The hello world application*

4 Applications

In order to demonstrate the usefulness of the presented approach, we work out three more illustrating examples. We describe a calculator, an editor and a combination of these two.

4.1 The Calculator

We take a look at a simple desk-calculator (see Fig. 4). For this calculator we define a state (cf. Sect. 2.3) consisting of the actual value of the display and an accumulator function.

```

type CalcState = Var (Int, Int -> Int)

```

The function `calculator` initializes the state, opens the window and resets the display.

```

main      = startProg calculator
calculator = do { st <- newState (0, id)
                ; openWindow [Title "Calculator"] (calc st)
                ; setText "display" "0"
                }

```

The user interface is built using an entry widget and a matrix of buttons for the keypad. Whenever the user presses a digit, it is displayed and the value component of the state is updated. When an operator is pressed, the display is reset and the accumulator function is modified. After pressing the '=' button, the calculator evaluates the accumulator function.

The code listed below implements the calculator completely.

```

calc :: CalcState -> Widget
calc st = disp ^-^ keys where
  disp = entryW [Name "display", Relief Sunken, Width 12]
  keys = matrix 4 (map wid [ '1', '2', '3', '+',
                            '4', '5', '6', '-',
                            '7', '8', '9', '*',
                            'C', '0', '=', '/'
                            ])
  wid c = buttonW [Text [c], Command (lift (cmd c)), Width 3]

```

```

lift f = do { (disp, accu) <- readState st
              ; (disp',accu') <- (result . f) (disp, accu)
              ; setText "display" (show disp')
              ; writeState st (disp',accu')
            }

cmd `C` (d,a) = (0, id)
cmd `=` (d,a) = (a d, const (a d))
cmd `c` (d,a) | isDigit c = (10*d + ord c - ord `0`, a)
               | otherwise = (0, ((char2op c).a) d)

char2op `+` = (+)      char2op `-` = (-)
char2op `*` = (*)      char2op `/` = (div)

```

The function `wid` maps each displayed character to a widget with a corresponding action. This action is defined by `lift`, which embeds the associated command into the `GUI` monad.

4.2 The Editor

The editor example illustrates the use of layout combinators, menus and text fields (see Fig. 4).

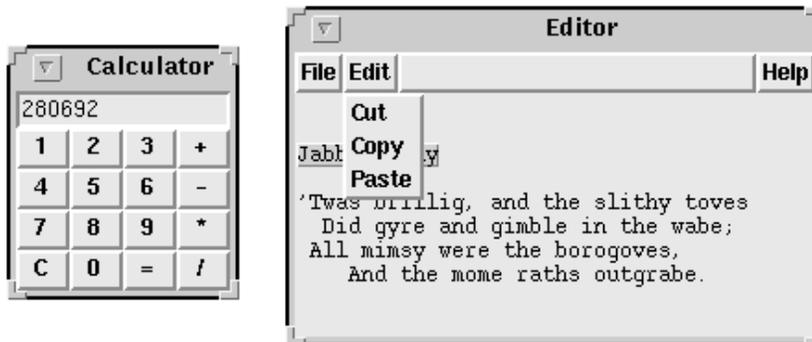


Fig. 4. *The calculator and the editor*

The editor state is a cut-copy-paste buffer, implemented by a sequence of strings. All other details concerning the state of the text widget, e.g., the actual position of the insertion cursor, are managed by the GUI.

```

type Buffer = Var [String]

main = startProg editor

```

```

editor = do { buf <- newState []
            ; openWindow [Title "Editor", Name "ed"] (edit buf)
            }

```

The menubar of the editor has three pull-down menus. The first two are ragged left, the last one is ragged right. By applying a configuration-map (<@), we raise the relief of the menubar. Below the menubar the textfield is placed.

```

edit :: Buffer -> Widget
edit buf =
  (fileM << editM << hfill << helpM) <@ ((Relief Raised):)
  ^_^
  textW [Name "text", Width 40, Height 10]
  where
    fileM = menuW [Text "File"]
              [cmdI "New" (warning "Really clear?" clear),
               cmdI "Quit" (warning "Really quit?"
                                closeWindow "ed")]

    editM = menuW [Text "Edit"]
              [cmdI "Cut" (cutE buf),
               cmdI "Copy" (copyE buf),
               cmdI "Paste" (pasteE buf)]

    helpM = menuW [Text "Help"] []

cmdI :: String -> Action -> MenuItem
cmdI t c = commandM [Text t, Command c]

```

If Cut, Copy or Paste is chosen from the Edit menu, the buffer is read or overwritten. The implementation of the corresponding functions is straightforward and therefore not listed here.

When New or Quit is chosen, a modal warning dialog opens. If the user presses No, the window is closed again; otherwise the confirmed action is performed.

```

warning :: String -> Action -> Action
warning s c =
  openWindow [Modal, Name "w", Title "Warning"] (l1 ^_^ b1 <+< b2)
  where l1 = labelW [Text s, Relief Raised, Width 20]
        b1 = buttonW [Text "No ", Command (closeWindow "w")]
        b2 = buttonW [Text "Yes", Command (do {closeWindow "w";c})]

```

4.3 Combining the Examples

To demonstrate the modularity and reuseability of GUI components, we combine the editor and the calculator. We let the calculator and the editor share the editor's buffer. The user now is able to copy the result of a calculation into the text editor and vice versa. To handle this extension we have to make some minor changes to the previously presented programs.

We add the menu Options to the editor, containing an item to invoke the calculator with the buffer.

```

edit´ buf =
  (fileM << editM << optM << hfill << helpM) <@ ((Relief Raised):)
  ^-^
  textW [Name "text", Width 40, Height 10]
  where
    optM = menuW [Text "Options"]
            [cmdI "Calculator ..." (calculator´ buf)]
    ...

```

The definition of the calculator is extended with the, now global, buffer. Furthermore, the calculator needs its own copy and paste commands. Therefore, we simply add a menubar to the calculator.

```

calculator´ buf =
  do { st <- newState (0, id)
      ; openWindow [Name "cc", Title "Calculator"] (calc´ st buf)
      ; setText "display" "0"
    }

calc´ st buf = (fileM << editM << hfill) ^-^ (calc st) where
  fileM = menuW [Text "File"]
              [cmdI "Quit" (closeWindow "cc")]
  editM = menuW [Text "Edit"]
              [cmdI "Copy" (copyC st buf),
               cmdI "Paste" (pasteC st buf)]

```

The Copy command writes the value of the display into the global buffer, the Paste command replaces the contents of the buffer by the displayed value. Their implementation is not shown here.

Note that we were able to reuse the original definition of `calc`. This nicely demonstrates how we can define modular building blocks for attractive user interfaces.

5 Implementation

This section discusses the runtime system, in particular the communication between the GUI library routines from Sect. 3 and the Tcl/Tk toolkit.

5.1 Tcl and Tk

The combination of Tcl and Tk [Ous94] provides a simple and comfortable programming system for developing small applications and graphical user interfaces.

Tcl. Tcl (tool command language) is an interpretive script language for controlling and extending applications. The syntax and semantics are close to C and the Unix shell. Tcl is an embeddable language, i.e., the language is in fact a library, designed to be linked together with other applications.

Tk. Tk is a toolkit for the X Window System [SG86] based on Tcl. It offers a set of widget commands for the creation of Motif-like user interfaces. All of the functionality of Tk-based applications is available through Tcl, i.e., evaluation of X events in Tk is done by invoking Tcl commands. The underlying event loops, call-back and display routines are all hidden away from the programmer.

5.2 Communication

It is surprising how little Tcl scripting is necessary to link a functional program to Tcl/Tk. In fact, the most simple, but already quite effective method is to run the program and Tcl/Tk as separate processes which are linked by a bidirectional pipe. Tcl/Tk sends events to the program (via stdin), which for its part returns its actions (on stdout) to Tcl/Tk.

Separating the GUI manager and the application in two processes poses two problems: which communication protocol should be used, and, where will information be stored?

We decide to store a minimal amount of GUI information in the functional program, viz. a call-back list. This list associates widget identifiers with actions. It suffices to communicate events by sending these identifiers from the GUI manager to the application. The identifier describes on which widget an event has happened. The event loop of the functional program then evaluates the corresponding command stored in the call-back list.

As Tcl/Tk is an interpretive language and the Tk operations are on a very high abstraction level, we represent actions that change the GUI as executable Tcl/Tk scripts.

However, not all user manipulation of the GUI triggers an event, e.g., typing text in a text field. So, information in the functional program is not always up to date. If we need the actual information from the GUI, a request is sent to the GUI manager: a Tcl/Tk script which, when evaluated, responds with the actual value. In any case, this strategy leads to synchronous communication.

Example: A trace of the editor. For the functional program, see Sect. 4.2.

First, the application must have the user interface set up. It generates the following script:

```
▷  wm title . "Editor"
▷  pack [frame .@1 ] -in .
▷  pack [frame .@2 ] -in .@1 -fill x
...
▷  pack [menubutton .@5 -relief raised \
▷      -menu .@5.m -text "Edit"] -in .@2 -side left
▷  menu .@5.m
▷  .@5.m add command -label "Cut" -command {communicate "C .@5m0"}
▷  .@5.m add command -label "Copy" -command {communicate "C .@5m1"}
...
▷  pack [text .text -width 40 -height 10 ] -in .@1 -fill x
▷  EOM
```

Tcl/Tk interprets these commands and displays window and widgets. The `EOM` terminates the action.

Second, suppose that the user has entered some text and marked the word “Jabberwocky” — this does not generate an event. When the user chooses the menu item Copy, the GUI communicates the event

```
< C .@5m1
```

In order to fill its buffer, the application must get the actual value of the marked text from the GUI. The application therefore sends the action

```
> write_event [.text .get sel.first sel.last]
```

which, when evaluated, responds with the actual marked part, namely

```
< Jabberwocky
```

The application reads this string and updates the buffer. Hereafter, it sends

```
> EOM
```

to terminate the (inter)action.

5.3 The GUI

The GUI starts with evaluating the `init` procedure. It uses the Tcl command `open` to create the Gofer application `$prog` as its subprocess. Once it is created, interaction is possible via the pipe `channel`, using the commands `puts` and `gets`.

```
proc init {prog} {
    global channel
    set channel [open "|$prog" r+]; read_actions
}
```

Calling the read-eval loop `read_actions` is the last command of the `init` procedure. This procedure reads and evaluates actions, i.e., Tcl/Tk scripts, until the message `EOM` is read. Obviously, the set-up script of the GUI must be the first message from the application.

```
proc read_actions {} {
    global channel
    set act [gets $channel]
    while {"$act" != "EOM"} {eval "$act"; set act [gets $channel]}
}
```

It then gives control to the Tcl/Tk event handler.

Whenever the user presses the button, the Tcl/Tk procedure `communicate` is invoked, which writes events into the pipe and starts the read-eval loop of Tcl/Tk again, to interpret the replied actions of the application.

```
proc communicate {msg} {
    write_event $msg; read_actions
}

proc write_event {msg} {
    global channel
    puts $channel $msg; flush $channel
}
```

This script is all you have to write in Tcl.

6 Related Work

The integration of referential transparent I/O in functional languages has been studied for a long time. We consider here only research to integrate GUIs in functional languages in a referential transparent way.

Among the first who integrated a GUI in a functional language was S. Singh [Sin92]. This approach already used a back to back arrangement, i.e., the application, written in Miranda, and the GUI, written in C, communicate over pipes. Both sides of the pipe supply primitive interpreters. However, (nearly) no abstraction is used: the graphical library XView is too low level – thus requiring a lot of ‘non-portable’ implementation work – and the GUI is implemented as an ordinary state variable. Thus, it is not asserted that the GUI representation in Miranda is used single threaded, whereas our monadic framework ensures this.

An alternative solution to this last problem was proposed by the Clean group [AvGP93], in the form of unique types. A unique type is guaranteed to be used only single threaded. Thus, the danger of copying GUIs is avoided. They developed a library with primitives similar to those of the Macintosh Toolbox, and implemented this GUI as a part of their runtime system.

A rather different approach was presented by M. Carlsson and T. Hallgren [CH93]. Instead of using the traditional widgets, they proposed fudgets as a functional equivalent. Fudgets may be thought as processes having two streams between them: one for high-level messages, connected to the application, and one for low-level X-messages. A button, for instance, is a process that, when pressed (low-level message), emits a click (high-level message). Fudget GUIs, based on streams, are programmed in a continuation style. Although this approach is rather nice, it has some shortcomings too. Problems mainly concern the inherent combination of stream based coupling and visualization of widgets (cf. [RS93]).

A work that also picks up Tcl/Tk as its GUI basics was already presented by D.C. Sinclair [Sin93]. He proposes to combine Haskell and Tcl/Tk using pipes, too. However, he does not hide Tcl/Tk behind the curtain — as we do — but uses a command-line oriented protocol between both processes.

Basically the same approach as Sinclair’s was pursued in our own previous work [SV94]. However, we did not have to modify the Tcl shell as he did. Furthermore, we showed how structured functional code is achievable using this approach. The disadvantage, however, remains that we had to program the user interface itself in pure Tcl/Tk.

Lastly, Caml Light[PR94] was also linked to Tcl/Tk, but in a purely imperative way.

7 Conclusion

In this paper, we presented a referential transparent functional library to develop modular applications with a lightweight graphical user interface. We showed how abstraction mechanisms like monads and layout combinators can be used to realize such a library. The here presented solution enabled us to make a very

smooth and high-level interface to the application programmer — who can now fully concentrate on the core of his system instead of fuzzy details of window programming.

The resulting system, composed of the Gofer GUI library and the Tcl/Tk script is small, easily extendable, reasonably fast and, due to the robustness of Tcl/Tk, stable.

Furthermore, our technique does not require any changes in runtime systems, as many other approaches do. Moreover, the needed I/O primitives are limited to the bare basics: reading and writing characters. Thus, implementation is not limited to Gofer, but can be adapted to any functional language, as long as it provides textual (monadic) I/O.

Alternatively, to improve efficiency, one could choose to embed Tcl/Tk into the Gofer runtime system. Since Tcl/Tk is particularly suited for such, this requires only a few local changes in the Gofer runtime system. For reasons of simplicity and clarity, we decided to present the transparent solution via pipes.

Acknowledgements. We would like to thank Erik Meijer for encouraging us to pursue this research; Klaus Achatz, Max Geerling, Magnus Carlsson and Thomas Hallgren for their helpful comments.

References

- [AvGP93] P.M. Achten, J.H.G. van Groningen, and M.J. Plasmeijer. High-level specification of I/O in functional languages. In *Glasgow Workshop on Functional Programming 1992*. Springer Verlag, 1993.
- [BW85] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer-Verlag, 1985.
- [CH93] M. Carlsson and Th. Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *Conference on Functional Programming and Computer Architecture*. ACM Press, 1993.
- [HPJW92] P. Hudak, S.L. Peyton Jones, and Ph. Wadler (eds.). Report on the programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.
- [JD93] M.P. Jones and L. Duponcheel. Composing monads. Research Report RR-1004, Yale University, December 1993.
- [Jon93a] M. Jones. *Release notes for Gofer 2.28*, 1993. Included as part of the standard Gofer distribution.
- [Jon93b] M.P. Jones. *An introduction to Gofer (draft)*, 1993.
- [KL94] D.J. King and J. Launchbury. Lazy depth-first search and linear graph algorithms in haskell. Technical report, University of Glasgow, 1994.
- [LPJ94] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. Technical report, University of Glasgow, November 1994.
- [Mog89] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, pages 14–23, Washington DC, 1989. IEEE.
- [NR94] R. Noble and C. Runciman. Functional languages and graphical user interfaces — a review and a case study. Available by ftp at <ftp.york.ac.uk>, February 1994.

- [Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, 1994.
- [P JW93] S.L. Peyton Jones and Ph. Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993.
- [PR94] F. Pessaux and F. Rouaix. The CamlTk interface — release beta2. Available by ftp at <ftp.inria.fr>, February 1994.
- [RS93] A. Reid and S. Singh. Implementing fudgets with standard widget sets. In *Glasgow Workshop on Functional Programming 1993*, 1993.
- [SG86] R.W. Scheiffler and J. Getty. The X window system. *ACM Transactions on Graphics*, 5(2), 1986.
- [Sin92] S. Singh. Using XView/X11 from Miranda. In *Glasgow Workshop on Functional Programming 1991*, Workshops in Computing. Springer Verlag, 1992.
- [Sin93] D.C. Sinclair. Graphical user interfaces for Haskell. In *Glasgow Workshop on Functional Programming*. Springer Verlag, 1993.
- [SV94] W. Schulte and T. Vullings. Linking reactive software to the X-Window system. Technical Report 94-14, Universität Ulm, December 1994.
- [Wad90] Ph. Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990.
- [Wad92] Ph. Wadler. The essence of functional programming. In *ACM Principles of Programming Languages*, 1992.
- [Wad94] Ph. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, pages 39–56, January 1994.