An Introduction to TkGofer

Ton Vullinghs, Wolfram Schulte, Thilo Schwinn

Spring 1996
Universität Ulm
Fakultät für Informatik
Germany

Contents

Preface

This report is an introduction to TkGofer. TkGofer is a library of functions for writing graphical user interfaces (GUIs) in the pure functional programming language Gofer. The library provides a convenient, abstract and high-level way to write window-oriented applications. The implementation rests on modern concepts like monads and constructor classes.

The main goal of this report is to illustrate the way in which you write GUIs in TkGofer. All the provided widgets are introduced and explained by a set of illustrating examples. The last part of this manual lists the signatures of the user functions of the GUI library.

TkGofer is freely available. For more information please contact `ton@informatik.uni-ulm.de`

Acknowledgments Several people have contributed their ideas and suggestions. Special thanks to Daniel Tuijnman, who designed and implemented substantial parts of earlier versions of the library. Furthermore we thank Erik Meijer and Klaus Achatz for their encouraging and helpful comments.

Chapter 1 Introduction

Functional programming languages offer many advantages to programmers. Using functional languages often results in faster development times and shorter code compared with imperative languages. Furthermore, reasoning about and reusing programs is easier. Recent research in the field of functional programming resulted in new concepts such as monads to tame the imperative aspects of I/O and state [Wad90], and constructor classes to deal with higher order polymorphism [Jon95].

Today, the specification of graphical user interfaces (GUIs) is an essential part of any realization of interactive systems. To avoid multi-paradigm programming, it is an obvious idea to incorporate GUI programming in functional languages. Due to intrinsic state-based properties of GUIs, monads are an obvious and natural choice for their implementation. Other people already presented alternative solutions to this problem, see for example [AvGP93, CH93, NR95].

This document describes the TkGofer GUI Library, an extension of the functional language Gofer, based on the graphical user interface toolkit Tcl/Tk. The main goal of the document is to explain how to to write programs in TkGofer and to give a brief description of the implementation of the system. Since all the GUI functions are abstractions of Tcl/Tk procedure calls, it is useful but not necessary to know a little about Tk. The best way to get some insight in the ins and outs of Tcl/Tk is by reading John Ousterhout's book 'Tcl and the Tk Toolkit', published by Addison Wesley in 1994. For a good introduction to functional programming we refer to [BW89]. The release notes and reference manual included in the standard Gofer distribution will tell you all the details about functional programming in Gofer [Jon93a, Jon93b].

1.1 Starting TkGofer

The TkGofer interpreter looks and behaves exactly the same as the standard gofer interpreter. TkGofer starts with loading the file `tk.prelude`. It is an extension of the `cc.prelude` and contains all the standard definitions you will need to write GUI programs in Gofer. You can start TkGofer by entering `tkgofer`, after which your display will show something like:

```
Gofer Version 2.30a  Copyright (c) Mark P Jones 1991-1994
Reading script file "tk.prelude":

Gofer session for:
tk.prelude
Type :? for help
```

The command `:?` will give you an overview of the available interpreter commands.

1.2 The First TkGofer Program

Figure 1.1 shows one of the simplest applications you may possibly write in TkGofer. It shows an entry and a button widget. The entry displays an integer. The value of this integer is incremented when the button is pressed.

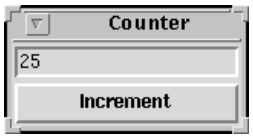

Figure 1.1: *A simple adder*

The program below shows all you have to write in TkGofer to implement the counter. Simply type 'adder' to let the example run.

```
adder :: IO ()
adder = start (
  window [title "Counter"]                      `bind` \w ->
  entry  [initValue 0] w                        `bind` \e ->
  button [text "Increment", command (incr e)] w `bind` \b ->
  pack (e ^-^ b)
  )

incr :: Entry Int -> GUI ()
incr e = getValue e `bind` (\x -> setValue e (x+1))
```

The function `adder` implements the user interface of the application. It creates a window, and two window items. The items are combined horizontally, using the combinator `^-^`. This means that the label and the entry are placed above each other and are aligned in length. The function `bind` combines two monadic actions.

The function `incr` defines the event that happens when we press the button. The actual value of the entry field is read, incremented and written back to the display.

1.3 Notation

1.3.1 Gofer

All programs are written in Gofer. GUI datatypes and functions are defined in the `tk.prelude`, Most of the GUI-functions have a monadic type, i.e., they return a value of type `GUI a`. To bind together monadic functions, you may use the functions `bind` and `result`. We prefer however a more readable style, using the Gofer `do`-notation.

Using the `do`-notation we can rewrite the previous example in the following way:

```
adder :: IO ()
```

```
adder = start $
  do w <- window [title "Counter"]
     e <- entry  [initValue 0] w
     b <- button [text "Increment", command (incr e)] w
     pack (e ^-^ b)

incr :: Entry Int -> GUI ()
incr e = do x <- getValue e ; setValue e (x+1)
```

To avoid nested bracketing in large expressions we will often use the `$` operator for infix function application. `$` is right associative and has the lowest precedence.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

1.3.2 NuWeb

This document is written using NuWeb [BR89]. NuWeb is a very simple literate programming environment that works with any programming language and LaTeX. Using NuWeb it is possible to write documentation for multiple program source files in a single document. It runs very quickly and has some nice features for generating HTML-pages and index-tables.

All the examples included in this document are executable in TkGofer. They are automatically extracted from this document if you run NuWeb.

1.4 What About Tk?

Since Gofer essentially serves as a generator for Tcl/Tk statements, it might be interesting to take a look at the generated code. For this purpose, we added the command line toggle `x`. Simply type the interpreter command `:set +x` and rerun the previous example. Your output will look like the following:

```
[Initialize Tk (4.1 or higher)]
  window .@0
  wm title .@0 "Counter"
  entry .@0.@1
  .@0.@1 configure -textvariable ".@0.@1"
  set svar0 0
  global .@0.@1 ;set .@0.@1 $svar0
  button .@0.@2
  .@0.@2 configure -text "Increment"
  .@0.@2 configure -command {doEvent 0}
  frame  .@0.@1f
  pack .@0.@1f -in .@0
  raise .@0.@1f
  pack .@0.@1 -in .@0.@1f -si top -fi x
  raise .@0.@1
  pack .@0.@2 -in .@0.@1f -si top -fi x
  raise .@0.@2
[Tk is waiting for an event...]
```

This listing is the exact Tcl/Tk code Gofer sends to the Tcl interpreter. Especially if you are writing extensions to the library, or if an unexpected Tk error occurs, you can debug the generated code in this way. You can reset the toggle by `:set -x`.

1.5 Overview

The rest of this document describes the main concepts of writing graphical user interfaces in Gofer. In Chap. 2 we will discuss some general principles of GUI programming in TkGofer. Important aspects like creating and combining widgets are explained. Furthermore, we sketch the role of the `GUI` monad and we explain the TkGofer type and constructor classes.

Readers mainly interested in GUI programming may want to skip directly to Chap. 3. In this chapter, all the standard widgets are introduced and explained on the basis of some illustrating examples.

Type and constructor classes make it possible to write extensions to the standard library. How to write new widgets is described in Chap. 4.

Finally, the last chapter serves as a reference manual to TkGofer. In this chapter we give all the signatures of the user-functions of the `tk.prelude`.

Chapter 2 Concepts

This chapter explains the main concepts of functional GUI programming using TkGofer. For a detailed discussion of the implementation of TkGofer we refer to [Sch96, VSS96b, VTS95]. Significant parts of TkGofer rest on advanced concepts like monads and higher order polymorphism. We therefore assume that you already have some knowledge about monadic programming and type and constructor classes. Detailed discussions about these topics can be found in [Jon95, LPJ95, PJW93, Wad90, Wad95].

2.1 The `GUI` Monad

The most important datatype of TkGofer is the `GUI` monad. The monad is implemented as a combination of the state reader monad and the `IO` monad [JD93]. Values of type `GUI a` represent actions that have some side effect on the user interface and return a value of type `a`. The type `GUI ()` represents all void actions; i.e., actions which only have a side effect and do not return a proper value.

For example the function `incr` (see the example in Sect. 1.2) has type `Entry Int -> GUI ()`. It reads a value and updates the entry field, but does not return a value. An example of a non-void action is the function `button :: [Conf Button] -> Window -> GUI Button`. It constructs a button widget and returns an identifier for this button.

Two frequently used monadic functions are `seqs` and `binds`. `seqs` 'executes' a list of void actions. `binds` 'executes' a list of non-void actions and returns a list of results. The function `void` throws away the result of a monadic action, thus performing a cast from `m a` to `m ()`.

```
seqs  :: Monad m => [m ()] -> m ()
binds :: Monad m => [m a] -> m [a]
void  :: Monad m => m a -> m ()
```

Monads can be used to implement lazy state threads, too [LPJ94, LPJ95]. We use them to store global data. A mutable variable is manipulated with the functions:

```
newState   :: a -> GUI (Var a)              -- create a variable
readState  :: Var a -> GUI a                -- read a variable
writeState :: Var a -> a -> GUI ()          -- write a variable
modState   :: Var a -> (a -> a) -> GUI ()   -- read/apply/write
```

Applications of mutable variables are given in Sect. 3.4.3 and 3.8.

2.2 Starting the Eventloop

In Gofer an interactive (monadic) program must have type `IO()`. All GUI applications begin with the function `start`. This function initializes the communication with Tcl/Tk and sets up the eventloop. The eventloop can be interrupted using Ctrl-C or the function `quit`.

```
start :: GUI () -> IO ()
quit  :: GUI ()
```

The argument of `start` denotes the first action to perform. Typically, this first action will create the user interface.

2.3 Creating Widgets

The basic building blocks of a graphical user interface are widgets. A widget is a graphical entity with a particular appearance and behaviour. We distinguish between four kinds of widgets: toplevel-, window-, menu- and canvas-widgets. Widgets of kind toplevel serve as containers for other widgets. Examples are windows and menus. Window-widgets, like buttons and labels, may appear on a window. Menu-widgets may occur in a pull-down or pop-up menu. Examples are buttons and separators. Canvas-widgets like rectangles and circles, may be placed on a canvas.

The different widget kinds are identified by the data constructors `T, W, M` and `C`. Normally, these constructors are hidden using type synonyms. For example, the type `Button` is defined by:

```
data Button0 = Button0
type Button  = W Button0
```

The constructor `W` defines the button as a window item. All other widgets are defined in the same way. For each widget, the library offers a construction function, e.g.:

```
window    :: [Conf Window] -> GUI Window
entry     :: [Conf (Entry a)] -> Window -> GUI (Entry a)
cascade   :: [Conf Cascade] -> Menu -> GUI Cascade
cline     :: (Int,Int) -> (Int,Int) -> [Conf CLine] -> Canvas -> GUI CLine
```

Defining the external outline of individual widgets is done by giving appropriate values for the configuration options. Examples are the color of a widget, a displayed text or the dimensions of a widget. The possible configuration options are widget specific (see also Sect. 3.3.3). To constraint options to a specific class of widgets we introduce a hierarchy of type and constructor classes [Jon95, VSS96a]. We explain the widget hierarchy in Sect. 2.4.

The exact behaviour of toplevel widgets, window widgets, menu widgets and canvas widgets, is explained in the examples in Chap. 3.

2.4 The TkGofer Widget Hierarchy

Although a lot of differences between widgets exist, most properties that widgets may have are shared by some widgets or even by all widgets, e.g., the way in which they have to be accessed or the way in which we have to specify their outline. Type and constructor classes are used to express the common characteristics of a set of widgets.

2.4.1 The Implementation Hierarchy

As said before, we distinguish between four kinds of widgets: toplevel-, window-, menu- and canvas-widgets. To exploit the similarities among the widgets of a certain kind, the classes `TopLevel`, `WindowItem`, `MenuItem` and `CanvasItem` are introduced. All widgets are instances of the class `Widget`. The program below shows class definitions for `Widget`, `WindowItem` and `MenuItem`:

```
class Widget a
class WindowItem a
instance Widget (W a) => WindowItem (W a)
class MenuItem a
instance Widget (M a) => MenuItem (M a)
...
```

The functions defined in these classes deal with implementation aspects, like the generation of widgets, and are not further discussed here. The constructed hierarchy is called the implementation hierarchy (see Fig. 2.1).

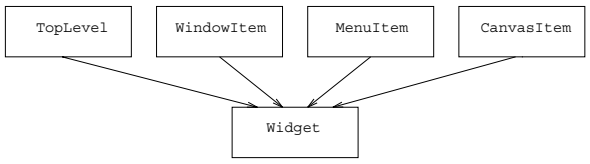

Figure 2.1: *The TkGofer implementation hierarchy*

2.4.2 The User Hierarchy

On top of the implementation hierarchy, we build other classes which are more oriented towards the application programmer. This leads to a hierarchy of classes as depicted in Fig. 2.2.

The basic class is called `HasConfigs`. In this class we define functions that apply to every widget, e.g., the function `cset` to update the configuration of a widget.

```
class Widget w => HasConfigs w where
    cset       :: w -> Conf w -> GUI ()
    ...
```

All other classes in the hierarchy are specializations of `HasConfigs`. `HasCommand`, for example, includes all widgets that additionally may be configured with a command. A typical instance is the datatype `Button`.

```
class HasText w => HasCommand w where
  command :: GUI () -> Conf w
  invoke  :: w -> GUI ()
  ...
```

An example of a constructor class is the class `HasInput`. In this class we group widgets that can handle user input. Instances are for example entry fields and texts (single and multiple line input). Widgets in this class are parameterized over the type of their input. This type should be an instance of the class `GUIValue` in which parse and unparse functions are defined to display values on the screen (see also Sect. 3.4.2). The class `HasInput` is listed below:

```
class (HasConfigs (c (w v)), GUIValue v) => HasInput c w v where
  getValue :: c (w v) -> GUI v
```

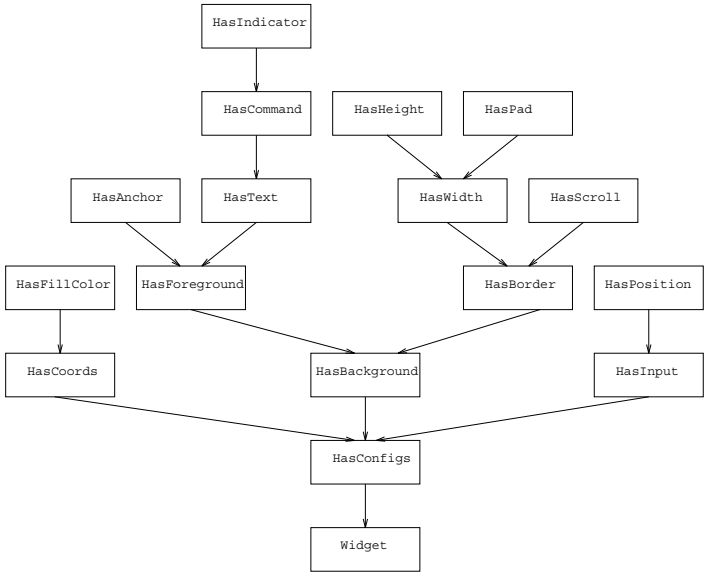

Figure 2.2: *The TkGofer user hierarchy*

```
setValue :: c (w v) -> v -> GUI ()
updValue :: (v -> v) -> c (w v) -> GUI ()
updValue f x = do i <- getValue x; setValue x (f i)
...
```

The class has three parameters: the constructor `c` ranges over the possible widget kinds (`T`, `W`, `M`, or `C`); `w` ranges over the constructors for widgets having an input value of type `v`.

A complete overview of all the classes and their member functions is given in Chap. 5. In Chap. 4 we will explain how to extend the widget hierarchy and how we can define new widgets.

2.5 Combining Widgets

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

```
(<<), (^^) :: (WindowItem (W a),WindowItem (W b)) => W a -> W b -> Frame
```

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 minimal 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. `flexible` is just an abbreviation for `fillXY . expand`.

```
fillX, fillY, fillXY, expand, flexible :: WindowItem (W a) => W a -> Frame
```

In Fig. 2.3 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.

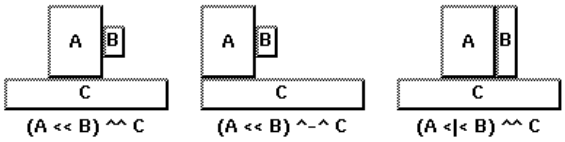

Figure 2.3: *Layout combinators and fill functions*

In Fig. 2.4 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 claimed and taken by B. Finally, in the last picture, the area is claimed, taken and divided by both A and B.

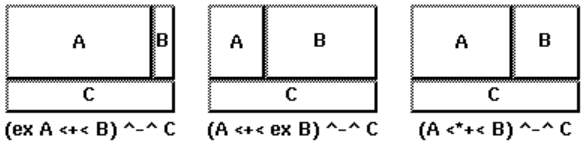

Figure 2.4: *Layout combinators and expand functions*

In these examples, `ex` abbreviates `expand`. The additional (associative) combinators combine the principles of sizing and positioning:

```
(<<), (<*<), (<-<), (<|<), (<+<), (<*-<), (<*|<), (<*+<),
(^^), (^*^), (^-^), (^|^), (^+^), (^*-^), (^*|^), (^*+^)
   (WindowItem (W a), WindowItem (W b)) => W a -> W b -> Frame
```

These combinators apply the same layout function on both arguments. For example `^-^` places two widgets above each other and aligns them in length, `<|<` place them next to each other, aligned in height. `+` is just the combination of `|` and `-`. Finally, `*` applies an expand operation on the right and left operand.

Chapter 3 Introducing Widgets

This chapter demonstrates the main techniques of writing GUIs in TkGofer. Similar to John Ousterhout's tour of the Tk Widgets ([Ous94], Chap. 16) we will briefly present the implemented widgets and describe how to create, configure and display them.

All the widgets are presented on the basis of some clarifying examples. To test the examples, you can read the generated files in the Gofer interpreter or simply load the project file `demo.p`. This project automatically includes the files:

```
label.gs
button.gs
message.gs
checkbutton.gs
setget.gs
radio.gs
scale.gs
entry.gs
entry_short.gs
calc.gs
listbox.gs
scrollbar.gs
octdec.gs
edit.gs
canvas.gs
histo.gs
```

All demos start with the name `ex_`*filename* (*filename* without the `.gs` extension). We proceed from the trivial 'Hello world' to more sophisticated programs like a desk calculator and a text editor.

3.1 Windows and Labels

Let's start with the famous 'hello world' example (see Fig. 3.1).

'Hello world' actually displays two widgets – a window and a label. In TkGofer, the implementation of this GUI looks like:

```
ex_label :: IO ()
```

Figure 3.1: *Hello World!*

```
ex_label = start $
  do w <- window [title "My Example"]
     l <- label [text "hello world", background "yellow", width 25] w
     pack l
```

A user interface may contain one or more windows. The window widget serves as a container for other widgets. The library offers functions to open, close and configure windows. The function `window` creates and opens (displays) a new window. It takes a list of configuration options as argument. In the above example, we configured the title of the window with the string "My Example". The actual position of the window is determined by the window manager of Tcl/Tk, or by the configuration options of the window.

The second widget is the label widget. A label is a widget that displays a string or a bitmap. The configuration options define the exact value of this string or bitmap. Other valid configuration options are for example the widget's background color or its dimensions. The last argument of the function `label` refers to the window in which the label has to be displayed.

Finally, the function `pack` displays the label. In general, `pack` is used to combine and display widgets that have to appear in the same window.

Since a combination of `window` and `pack` will occur very frequently in your programs, the prelude offers the function `openWindow` as an abbreviation for this. It takes a list of configuration options as argument. and a function which creates window widgets of type `W a`. `closeWindow` removes a window from your display.

```
openWindow  :: [Conf Window] -> (Window -> GUI (W a)) -> GUI ()
closeWindow :: Window -> GUI ()
```

Using this function, the example can be rewritten in a shorter way:

```
hello = (start . openWindow [title "My Example"])
  (label [text "hello world", background "yellow", width 25])
```

3.2 Messages

Message widgets are similar to labels except that they display multiline strings. A message automatically breaks a long string up into lines. The configuration function `aspect` controls the width/height ratio for the displayed text. Furthermore, with the function `justify`, we can center a text, or position it to the right or to the left.

```
ex_message :: IO ()
ex_message = start $
  do w   <- window [title "What´s the message?"]
     ms  <- binds
               [ message [ text msg, aspect (75*i), justify pos] w
               | pos  <- ["left","center", "right"], i <- [1..3]
               ]
     pack (matrix 3 ms)
  where msg = "the message widget displays and formats a text"
```

This example also demonstrates the function `matrix`. This layout function takes a number of columns and a list of widgets as its arguments and composes them in a row major fashion. Since the second argument is a list of widgets, all widgets must be of the same kind.

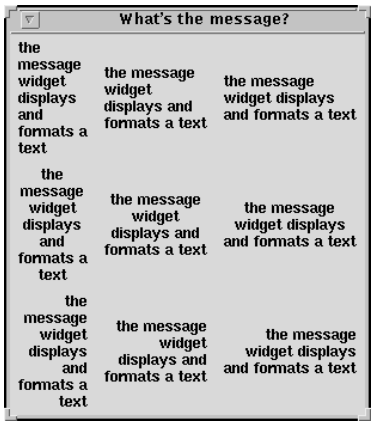

Figure 3.2: *Several variations of* `aspect` *and* `justify`

Build on `matrix` are the layout functions `horizontal` and `vertical`:

```
horizontal xs = matrix (length xs) xs
vertical   xs = matrix 1 xs
```

3.3 Buttons

In this section standard commandbuttons are introduced. Another variation are radiobuttons and checkbuttons. They have the same characteristics as commandbuttons, but additionally have some 'dynamic' feature.

3.3.1 Commandbuttons

Figure 3.3 shows an extension to the 'hello world' example. We added a button widget. A button is very similar to a label, except that it responds to the mouse. If the user moves the mouse pointer over the button, the button lights up to indicate that something will happen if the left mouse button is pressed — if a command option is specified, the argument of the function `command` is executed.

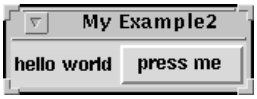

Figure 3.3: *Adding a button*

This argument has to be a void action, i.e., a function of type `GUI ()`. In the extended 'hello world' application, the program quits if the user presses the button.

```
ex_button :: IO ()
```

```
ex_button = start $
  do w <- window [title "My Example2"]
     l <- label  [text "hello world", background "yellow"] w
     b <- button [text "press me", command quit] w
     pack (l << b)
```

3.3.2 Checkbuttons

Checkbuttons have a binary state (true or false) which is set or unset, each time the user presses the button. Using the function `setValue` the user may give this widget a specific value, or, by using `getValue`, read the actual value of the button state. The following example illustrates the use of checkbuttons:

```
ex_checkbutton :: IO ()
ex_checkbutton = start $
  do w   <- window [title "Check this out!"]
     l1  <- label [text "The moon is made of cheese"] w
     cb1 <- checkbutton [ text "Press me", indicatorOn False
                        , indicatorColor "green", background "red"
                        ] w
     cb2 <- checkbutton [text "Wrong", width 8] w
     cset cb2 (command (pressed cb2))
     pack (cb1 ^-^ (l1 << cb2))

pressed :: Checkbutton -> GUI ()
pressed c =
  do b <- getValue c
     cset c (text (if b then "Right" else "Wrong"))
```

Figure 3.4: *Checkbuttons*

The first checkbutton (`cb1`) defines a red checkbutton, whose color changes to green when we press it (`indicatorColor "green"`). The function `indicatorOn` specifies that either a small indicator or the relief of the button (sunken or raised) informs us about the state of the button (`indicatorOn True` is default).

The second checkbutton (`cb2`) responds on a mouse event by calling the function `pressed`. `pressed` reads the state of the checkbutton and changes the text of the button correspondingly.

3.3.3 Configuring Widgets

Changing and reading the configuration options of already generated widgets is done using the functions `cset` and `cget`, respectively. Both functions take a widget and a configuration function as argument. `cset` and `cget` have the following signature:

```
cset :: HasConfigs a => a -> Conf a -> GUI ()
cget :: (HasConfigs a, GUIValue b) => a -> (b -> Conf a) -> GUI b
```

Configuration options are parameterized over the set of types they may apply on. For example the function `justify`, which is only allowed for message-widgets, has type `String -> Conf`

`Message`, whereas the function `background` has the restricted polymorphic type `HasBackground a => String -> Conf a`. This means that `background s` is a valid option for all widgets that are an instance of the class `HasBackground`. An example of `cset` and `cget` is the following 'reverse-button' function:

```
ex_setget :: IO ()
ex_setget =  start $
  do w <- window []
     b <- button [text "hello"] w
     cset b (command (rev b))
     pack b
  where
    rev b = do x <- cget b text; cset b (text (reverse x))
```

After pressing the button, the displayed text is reversed.

3.3.4 Sequentialization of Actions

A drawback of monadic programming is that it enforces a strong sequentialization of actions. In the previous program for example, we first generate a button, and then we apply the function `cset`. The obvious reason for this is that we cannot use the variable `b` before it is generated. There are however some tricks to solve the problem in some special situations.

Consider the function self, defined by:

```
self :: (a -> a -> b) -> a -> b
self f a = (f a) a
```

Since configuration options essentially are functions from widgets to options, we can apply `self` to abbreviate the sequence

```
x <- widget [] w; cset x (c x)
```

by

```
x <- widget [self (\x -> c x) ] w
```

`self` takes the generated widget as an argument for the configuration option. Applied to the `cset_cget` example we get:

```
b <- button [ text "hello", self (command . rev)]
```

3.3.5 Radiobuttons

Radiobutton widgets are used to select one of several mutually exclusive options. The buttons are controlled by one (abstract) widget — the radio. For radiobuttons and radios, the same operations are defined as for checkbuttons.

The next example (see Fig. 3.5) shows the use of radios and radiobuttons. The program simulates a very primitive trafficlight protocol.

```
ex_radio :: IO ()
ex_radio = start $
  do w <- window []
     (ls1, r1) <- traffic w
     (ls2, r2) <- traffic w
     seqs (control ls1 r1 r2 ++ control ls2 r2 r1)
     pack (vertical ls1 << vertical ls2)
  where
    traffic w =
      do bs <- binds [radiobutton [indicatorColor c] w
```

```
                  | c <- ["red", "yellow", "green"]
                  ]
      r  <- radio [initValue 1] bs
      result (bs, r)

control ls i j =
  [cset b (command $ do x <- getValue i; setValue j (2-x)) | b <- ls]
```

Figure 3.5: *A small trafficlight controller*

The function `radio` takes a list of radiobuttons as parameter and returns a controller for the group of buttons. The functions `setValue` and `getValue` are used to address the buttons of a radio. `setValue` takes an integer value, corresponding to the position of the radiobutton to set. Likewise, `getValue` returns the position of the actual selected button.

The function `traffic` creates one trafficlight. It returns a list of three buttons and the radio to control them. Initially, both trafficlights are yellow (`initValue 1`). The function `control` assigns a command to every button, which guarantees the exclusiveness of the two trafficlights.

3.4 Entries

An entry widget allows the user to type in and edit a one-line string. This string may represent any displayable type in a TkGofer program. Entries have some dynamic contents. Since entries are an instance of same class (viz. `HasInput`) as radio- and checkbuttons we may access them again using `getValue` and `setValue`.

The next example (see Fig. 3.6) implements a simple adder. When the user presses the enter button, the value of the entry field is increased.

```
ex_entry :: IO ()
ex_entry = start $
  do w <- window []
     e <- entry [initValue 0] w
     cset e (on return $ do x <- getValue e; setValue e (x+1))
     pack e
```

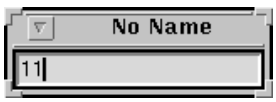

Figure 3.6: *A simple adder*

This example demonstrates the use of user defined events (`on .. do ..`). They correspond

to 'bindings' in Tcl/Tk. The first argument is some key or mouse event, the second argument defines the function that is called if the event occurs.

The function `initValue` initializes the contents of the entry. By the way, if you do not like writing your applications using the `do`-notation, you might like to write the previous example in 'dutch' style:

```
ex_entry_short :: IO ()
ex_entry_short =
  (start . openWindow [])
    (entry [self $ on return . updValue (1+), initValue 0])
```

`updValue` is an abbreviation for reading a value, applying a function to it, and writing the resulting value.

3.4.1 Typed Contents

An important feature of entry widgets (and also of other widgets with input) is the fact that they are typed over their contents. Entries, displaying integers, have type `Entry Int`. Entries displaying booleans have type `Entry Bool`, etc. If we want to assign a string to an integer input field, we get the following error message:

```
setValue e "hello"

*** expression     : setValue e "hello"
*** term           : e
*** type           : W (Entry0 Int)
*** does not match : a (b [Char])
```

`W (Entry0 Int)` is the derived type for `e`. `W (Entry0 a)` may be abbreviated by the type synonym `Entry a`.

Remember that gofer needs enough information to derive the exact type of a widget. The following program cannot be typed correctly:

```
type_error = start $
  do w <- window []
     e <- entry [] w
     pack e

ERROR "entry_error.gs" (line 1): Unresolved top-level overloading
*** Binding             : type_error
*** Inferred type       : IO ()
*** Outstanding context : Widget W (Entry0 _26)
```

To solve this problem we can explicitly type the widget, for example by replacing the last line by:

```
pack (e :: Entry Int)
```

or we can give some hints by providing an initial value for the entry field. In most applications however, a widget occurs within a special context — this context determines the type of the widget.

3.4.2 Reading and Writing Arbitrary Values

The previous section showed that some widgets are parameterized over their contents of some type `a`. Values of type `a` are printed on your display and can be read (user-input).

Since Tcl/Tk only deals with strings, we have to convert every displayed type in our application to string. Likewise, we have to parse strings if we want to use the input. The class `GUIValue` defines

parse and unparse functions for any value that may be displayed at the GUI. If parsing fails we open a standard error dialog.

The following example shows a decimal-octal converter after entering an invalid input value (see Fig. 3.7).

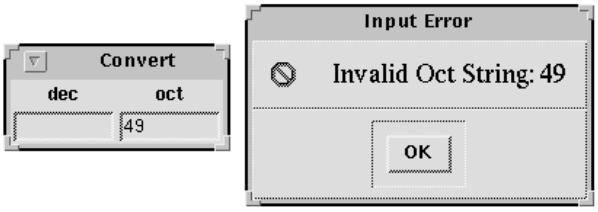

Figure 3.7: *A decimal octal converter*

We define `Octal` as an instance of the class `GUIValue`. We have to write an instance for the functions `tk_defaultValue` and `tk_convert`. `tk_defaultValue` denotes the value we have to return in case an input error occurs. `tk_convert` specifies the parse routine for the type `Oct`. Unparsing is defined by the function `show` as a default. Therefore, we have to write an instance of the class `Text` for the type `Oct`.

```
data Oct = Oct Int

instance GUIValue Oct where
  tk_defaultValue = Oct 0
  tk_convert s | all (flip elem "01234567") s = Tk_Ok (Oct (numval s))
  tk_convert s | otherwise = Tk_Err ("Invalid Oct String: " ++ s)
instance Text Oct where
  showsPrec d (Oct x) = shows x
```

The application itself consists of two entry fields. The first entry has type `Entry Int`, the second one `Entry Oct`. Each time a value is entered in the decimal entry field the octal one displays the converted value and vice versa.

```
ex_octdec = (start . openWindow [title "Convert"]) conv where
  conv w =
    do (f1,e1) <- input w "dec"
       (f2,e2) <- input w "oct"
       doconv (\n -> Oct (fromTo 10 8 n)) e1 e2
       doconv (\(Oct n) -> fromTo 8 10 n) e2 e1
       result (f1 << f2)

  input w s =
    do l <- label [text s] w
       e <- entry [width 9] w
       result ((l ^-^ e),e)

  doconv f a b =
    cset a (on return (do {x <- getValue a; setValue b (f x)}))

  fromTo n m = foldr (\a b -> b*n + a) 0 . digits m
    where digits j n = map (`mod` j) ((takeWhile (>0) . iterate (`div` j)) n)
```

3.4.3 State and GUI: The Calculator

Another demonstration of entries and buttons is the desk-calculator example. This example shows how to deal with a global state. So far, we only met examples that did not use a global state. Actions only had some (local) side-effect on the GUI. In this example, we will need actions that use mutable variables (cf 2.1).

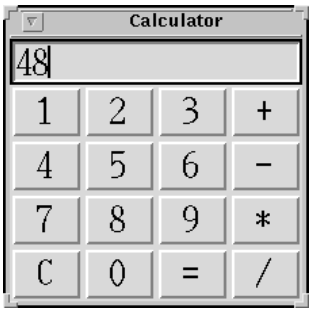

Figure 3.8: *The calculator*

Take a look at Fig. 3.8 and imagine what should happen if the user presses the button '+'. The calculator has to read the actual value of the display and apply '+' to it. The calculator has to keep this accumulator function in its memory, till the user has entered a new number and pressed another operator key.

Since every command has the type `GUI ()`, we cannot return the updated 'memory' as the result of a button press. The simplest way to solve this problem is to use mutable variables. We implement the calculator's memory by a mutable variable which contains the displayed value and the value of the accumulator function. The definition of the calculator state and GUI is given below.

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

ex_calc :: IO ()
ex_calc = start $
  do st <- newState (0, id)
     w <- windowDefault [title "Calculator"] [font "12x24"]
     c <- calc st w
     pack c

calc :: Var CalcState -> Window -> GUI Frame
calc st w =
  let disp   = entry [relief "sunken", width 12, initValue 0] w

      keys e = map (cmd e) [ ´1´, ´2´, ´3´, ´+´,
                             ´4´, ´5´, ´6´, ´-´,
                             ´7´, ´8´, ´9´, ´*´,
                             ´C´, ´0´, ´=´, ´/´
                           ]

      cmd e c = button [text [c], command (next e (action c)), width 2] w

      next e f = do (disp, accu)  <- readState st
                    let (disp´,accu´) = f (disp, accu)
                    setValue e disp´
                    writeState st (disp´,accu´)
```

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

    char2op '+' = (+)
    char2op '-' = (-)
    char2op '*' = (*)
    char2op '/' = \x y -> if y == 0 then 99999999 else x `div` y

in do e <- disp
      k <- binds (keys e)
      result (e ^-^ matrix 4 k)
```

The function `calc` initializes the state and opens the window. The function `windowDefault` applies the second list of configuration options to every widget in the GUI.

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.

3.5 Scales

A scale widget is a widget that displays an integer value and allows users to edit this value by dragging a slider. We have two functions to generate scales, `hscale` for horizontal scales and `vscale` for vertical scales. The range of values for the scale is specified by the function `scaleRange`. To display tickmarks next to a scale, we use `tickInterval`.

If the command option is specified, each time the value of the scale changes, the command is executed. The example below (see Fig. 3.9) shows the use of scales.

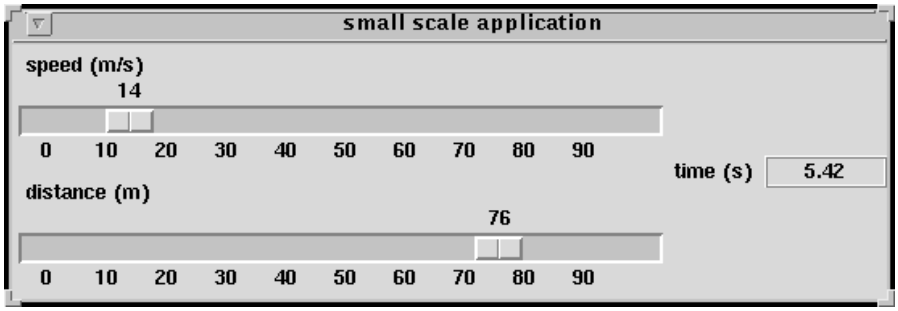

Figure 3.9: *Calculating the total time*

The two scales represent an indicator for speed and distance. When the user moves the scales, the values are increased or decreased. The corresponding trip duration is recalculated and displayed.

```
ex_scale :: IO ()
ex_scale = start $
  do w <- window [title "small scale application"]
```

```
    s1 <- makeScale "speed (m/s)" w
    s2 <- makeScale "distance (m)" w
    l1 <- label [text "time (s) "] w
    l2 <- label [width 10, relief "ridge"] w
    setCommands s1 s2 l2
    pack ((s1 ^-^ s2) <|< (l1 <*-< l2))
  where
    makeScale lab win =
      hscale [ scaleRange (0, 99)
             , tickInterval 10
             , text lab
             , height 400
             ] win
    setCommands s1 s2 l2 =
      let slide = do v <- getValue s1
                     d <- getValue s2
                     cset l2 ((text . take 4 . time d) v)
          time d 0 = "0.0"
          time d v = show ((fromInteger d / fromInteger v) :: Float)
      in do cset s1 (command slide)
            cset s2 (command slide)
```

3.6 Listboxes

A listbox is a widget that displays a collection of elements and allows the user to select one or more of them. Also listboxes are parameterized over the type of their contents.

The example shows two listboxes (see Fig. 3.10). The left one displays strings, the right one displays integers. In the right listbox we have marked four elements.

```
ex_listbox :: IO ()
ex_listbox = start $
  do w   <- window []
     l1  <- label [text "Strings"] w
     l2  <- label [text "Integers"] w
     lb1 <- listbox [initValue (part 3 ['A'..'Z'])] w
     lb2 <- listbox [initValue [1..8], multipleSelect True] w
     pack ((l1 ^-^ lb1) <|< (l2 ^-^ lb2))
  where part n = map (take n) . takeWhile (not . null) . iterate (drop n)
```

The type checker will derive that `lb1` has type `Listbox [String]` and `lb2` has type `Listbox [Int]`.

To switch between the two select modi (single select, to select only one element from the list and multiple select to select a set of elements) we use the function `multipleSelect`. Selections are made using the mouse. To select two or more non-consecutive elements, we can use the control key to fix the first selections.

3.7 Scrolling Widgets

Scrollbars control the view in other widgets. Therefore, a scrollbar is always associated with another widget. Scrollbars can be generated by the functions `vscroll` and `hscroll`. Both functions have the same signature. The first argument is a list of configuration options and the second argument refers to the widget to scroll.

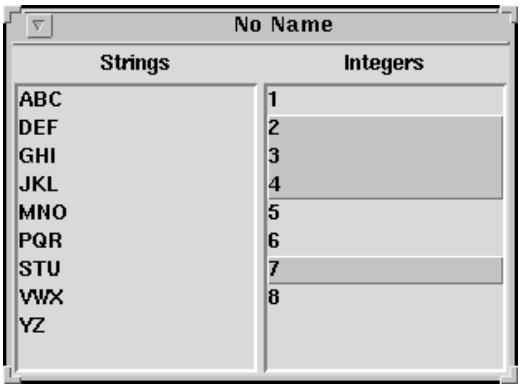

Figure 3.10: *Two listboxes, the right one with four marked elements*

Figure 3.11: *Scrolling and selecting*

Figure 3.11 shows a listbox and an entry. Both widgets are associated with a scrollbar. If the user selects a value in the listbox, the value is automatically displayed in the entry field.

```
ex_scrollbar :: IO ()
ex_scrollbar = start $
  do w <- window [title "select"]
     (e,f1) <- scrollEntry w
     (l,f2) <- scrollListbox w
     cset e (on return (readEntry l e))
     cset l (on (doubleClick 1) (writeEntry l e))
     focus e
     pack (f2 ^-^ f1)
  where
    scrollEntry w =
      do e <- entry [initValue ""] w
         s <- hscroll [] e
         result (e, e ^-^ s)
    scrollListbox w =
      do l <- listbox [] w
         s1 <- hscroll [] l
         s2 <- vscroll [] l
         result (l, (l ^-^ s1) <|< s2)
```

```
readEntry l e =
  do x <- getValue e
     putEnd l [x]
writeEntry l e =
  do [x] <- getSelection l
     [a] <- getFromTo l x x
     setValue e a
```

The application `focus e` sets the input focus to the entry widget `e`. This ensures that all keystroke events will arrive at the entry field.

The function `writeEntry` shows some other features of listboxes (and editors, as we will see in the next section). To read the positions of the actual selected items, we use `getSelection`. To read the actual values of the elements on these positions we use the function `getFromTo`. `getFromTo` takes two positions as its arguments and returns all the elements within this range.

3.8 Editors and Menubars

The Edit widget displays one or more lines of texts and allows you to edit the text. Many default key- and mouse-bindings exist to browse a text (e.g. cursor keys). Since Editors and Listboxes both belong to the same class, we may use the same functions to access and modify the contents of the widget.

Two more advanced techniques that deal with texts are provided as well: marks and tags.

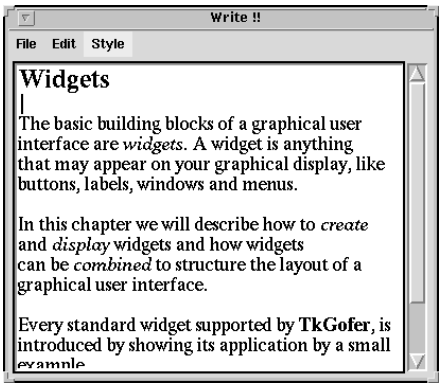

Figure 3.12:

We discuss the edit widget on the basis of a small editor example (see 3.12). Simultaneously, we will introduce the menu widget.

3.8.1 The Editor

We want to develop a small editor, with variable fonts and a simple cut-copy-paste buffer. To realize this, we will need a global state again. The editor state is a mutable variable, containing the contents of the buffer and the actual fontsize.

```
type State = Var (String,Int)  -- buffer, fontsize
```

```
ex_edit :: IO ()
ex_edit = start $ do
  st <- newState ("",18)
  w  <- window [title "Write !!"]
  e  <- edit [width 40, height 15, wrap True,
              background "white", font "times-roman18"] w
  s  <- vscroll []  e
  f  <- frame [borderWidth 4] (flexible e <|< s)
  bs <- menubar
          [("File", fileM e), ("Edit", editM e st), ("Style", styleM e st)] w
  pack (flexible (horizontal bs ^-^ flexible f))
```

The GUI consists of two main elements, a menubar and the edit-window. An edit-widget specific function is the function `wrap`; it determines whether a text should be broken into lines of words or lines of characters.

We use the frame widget to group and configure the edit widget and a scrollbar. Normally, widget-combinators generate frames and configure them with a default list of options. However, if we want to configure a frame explicitly, we may use the function `frame`. The function `menubar` is defined in the next section, It returns a list of menubuttons. We pack all the menubuttons horizontally using the function `horizontal`.

3.8.2 Menus and Menuitems

The menu widget can be used to implement pulldown menus, cascading menus and pop-up menus. A menu is a toplevel widget that contains a number of menu-items, arranged in a column. Possible items are buttons, radiobuttons, checkbuttons and cascade-menubuttons. They behave exactly the same as the corresponding window items. Furthermore, the separator widget just displays a horizontal line for decoration.

A pulldown menu is a menubutton with an associated menu. When the user presses the menu button, the menu is posted directly underneath the button.

The general pattern for creating pulldown menus is the following:

```
mb <- menubutton configs window    -- create menubutton
m  <- menu configs mb              -- create associated menu
b1 <- mbutton configs m            -- create menu items
b2 <- mbutton configs m            --
...
pack mb                            -- display menubutton
```

Notice that menu items are not packed. They are automatically displayed if the menu is posted. They are displayed in the order in which they were created.

A menubar is a vertical bar of menubuttons. The next code-fragment of the editor example shows a possible definition for the function `menubar`. It takes a list of tuples of strings and menu-items and associates every list of menu-items with a menubutton.

```
menubar :: [(String, Menu -> [GUI ()])] -> Window -> GUI [Menubutton]
menubar xs w =
 let (ss,fs) = unzip xs
 in do bs <- binds [menubutton [text s] w | s <- ss]
       ms <- binds [menu [] b | b <- bs]
       (seqs . concat) [f m | (f,m) <- zip fs ms]
       result bs

cmd :: (String, GUI ()) -> Menu -> GUI ()
cmd (s,c) m = void (mbutton [text s, command c] m)
```

```
fileM :: Edit -> Menu -> [GUI ()]
fileM e m =
  [cmd s m | s <- [("New", doNew), ("Quit", doQuit)]]
  where doNew  = warning "Really Clear?" (setValue e "")
        doQuit = warning "Really Quit?" quit
```

The function `cmd` creates a commandbutton menuitem. This button behaves exactly the same as the standard button widget. Since we do not have to refer to this widget any longer, we apply the function `void` to nullify the resulting widget.

The first pulldown menu of the editor is implemented by `fileM`. It creates two menu items, i.e., a `new` and a `quit` button. Both commands buttons open a warning dialog if they are pressed.

```
warning :: String -> GUI () -> GUI ()
warning msg yes = do
   w  <- windowDefault [title "Warning"][font "helvetica18"]
   m  <- message [text msg, relief "ridge"] w
   b1 <- button [text " Yes ", command (closeWindow w `seq` yes)] w
   b2 <- button [text " No " , command (closeWindow w)] w
   f  <- frame [borderWidth 2, relief "ridge"] (b1 << b2)
   focus b2
   pack (m ^*+^ f)
```

The second argument of the warning dialog is the action to perform if the Yes-button is pressed. If the user presses the No-button, the dialog is closed.

3.8.3 Marks

Text operations often refer to some particular place in the text. For example an append action refers to the end-position of the actual input, whereas an insert action refers to the actual cursor position. Using marks we can read the actual value of the mouse cursor (`mouseMark`), the insertion cursor (`insMark`) and the end of the text (`endMark`). The function `getMark` reads the actual value of the mark, `setMark` updates this value.

In the definition of `doPaste` we find an application of marks; we want to paste the text at the actual cursor position.

```
editM :: Edit -> State -> Menu -> [GUI ()]
editM e st m =
  cset e (onXY (click 3) (\xy -> popup xy m)) :
  [cmd s m | s <- [("cut", doCut), ("copy", doCopy), ("paste", doPaste)]]
  where doCut = selectionExists e ==> do
          ([p,q],t) <- getMarkedPart e
          delFromTo e p q
          modState st (\(_,i) -> (t,i))
        doCopy = do
          (_,t) <- getMarkedPart e
          modState st (\(_,i) -> (t,i))
        doPaste = do
          (t,_) <- readState st
          p <- getMark e insMark
          putPos e p t

selectionExists :: Edit -> GUI Bool
selectionExists e = do
  ps <- getSelection e
  result (ps /= [])
```

The edit menu contains three command buttons to cut, copy and paste pieces of text. The corresponding actions read and write the cut copy paste buffer. The first line of the body of the function `doEdit` defines an extra mouse binding for the edit-window. When the user presses the right mouse button, the cut-copy-paste menu is popped up directly underneath the mouse cursor. The second line actually defines the menu-items.

In the definition of `doCut` we see an application of the assert operator `==>`. It takes a (monadic) conditional action as it first operand and only evaluates its second argument if the condition evaluates to true:

```
(==>) :: GUI Bool -> GUI () -> GUI ()
```

(In fact, this is almost the same as the monadic if operation does [Jon93b], but since we cannot define a `zero` operation for the `GUI` monad, we redefined this operation).

The function `selectionExists` uses the library function `getSelection`. It returns the range of the actual selection.

3.8.4 Tags

Tags are used to change the appearance of a particular piece of text, e.g., change its color or font. A tag represents a piece of text, identified by a list of positions that may be configured just like any other widget. The library offers operations for creating and deleting tags (`tag`, `putEndTag`, `putPosTag` and `delTag`). Furthermore, since a text fragment can be a part of more than one tag, an operation `lowerTag` exists, to modify the stacking order of tags.

In the editor-example, we use tags to change the font of a marked text part. The function `setf` first checks whether a selection exists or not, and if so, it will read the coordinates of the marked part and create a tag for this range with the desired font and fontsize.

```
styleM :: Edit -> State -> Menu -> [GUI ()]
styleM e st m = fonts ++ [void (separator m), subm]
  where fonts =
          [cmd (s ,setf ("times-"++s)) m | s <- ["roman", "bold", "italic"]]
        subm = do
          mb <- cascade [text "font size"] m
          m  <- menu [] mb
          bs <- binds [mradiobutton
                  [text (show n), command (modState st (\(t,_) -> (t,n)))] m
                      | n <- [8,10..24]]
          void (radio [] bs)
        setf s = selectionExists e ==> do
          (ps,_) <- getMarkedPart e
          (_,n)  <- readState st
          void (tag ps [font (s++show n)] e)

getMarkedPart :: Edit -> GUI ([(Int,Int)],String)
getMarkedPart e = do
  ps <- getSelection e
  case ps of [a,b]     -> do tx <- getFromTo e a b
                             result (ps,tx)
             otherwise -> result ([],"")
```

3.9 Canvas and Canvas Items

A canvas is a widget that displays a drawing surface on which you can place various items. TkGofer currently supports rectangles, ovals, lines, texts and bitmaps. To display canvas items, we first

have to create a canvas. Every canvas item takes the canvas it should appear on as parameter.

If a canvas item is created, it is automatically displayed at the position specified in the first parameters. Items can be manipulated by changing the configuration options and coordinates.

Figure 3.13:

The example (see Fig. 3.13) shows how to create and modify canvas items.

```
ex_canvas :: IO ()
ex_canvas = start $ do
  do w <- window [title "Move It!"]
     c <- canvas [background "white", width 200, height 200] w
     r <- crect (10,10) (50,50) opts c
     l <- cline (70,70) (120,120) opts c
     o <- coval (150,150) (200,200) opts c
     t <- ctext (10,130) [ text "hello world", moveIt, raiseIt] c
     pack c

opts :: HasFillColor (C a) => [Conf (C a)]
opts = [penWidth 3, penColor "red", fillColor "yellow", moveIt, raiseIt]

moveIt :: HasCoords (C a) => Conf (C a)
moveIt = self (onxy (motion 1) . moveIt´) where
  moveIt´ w (x,y) =
     do ((x´,y´):ys) <- getCoords w
        moveObject w (x - x´, y -y´)

raiseIt :: HasCoords (C a) => Conf (C a)
raiseIt = self (on (click 1) . raiseObject)
```

The function `moveIt` is called if we want to drag an item to another position. The function `raiseIt` puts an item on top of another item if two items overlap.

3.10 Drawing a Histogram

We conclude the description of the standard TkGofer widget set with a small example to illustrate the expressive power of functional GUI programming. It combines a small GUI and a function to calculate a histogram for some given list of integers.

```
ex_histo :: IO ()
ex_histo = start $ do
  w <- window [title "Histogram"]
  c <- canvas [width (xmax +10), height (ymax + 10)] w
  e <- entry  [self (on return . draw c)] w
  pack (c ^-^ e)

draw :: Canvas -> Entry String -> GUI ()
draw c e = do
  clearCanvas c
  v <- getValue e
  seqs [ (void . crect (x1,y1) (x2,y2) [fillColor "cyan"]) c
       | (x1,y1,x2,y2) <- (bars . map numval . words) v
       ]

bars :: [Int] -> [(Int,Int,Int,Int)]
bars bs =
  let yunit  = fromInteger ymax / fromInteger (maximum bs)
      xunit  = xmax / length bs
      hght i = ymax + 5 - truncate (fromInteger i * yunit )
  in [(x+5, hght y, x+xunit+3, ymax) | (x,y) <- zip [0,xunit..] bs]

xmax = 150
ymax = 100
```

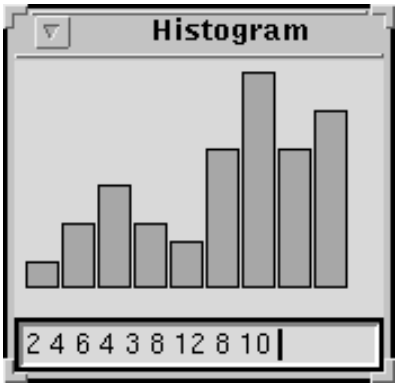

Figure 3.14:

If the user presses the enter key, the numbers displayed in the entry field are converted to integers and displayed as rectangles.

Chapter 4 Defining New Widgets

Very often, a user interface is composed of building blocks, which are reused a number of times. In this chapter we describe the steps you should take, to write new widgets as a combination of other, more primitive widgets. The examples in this chapter are automatically loaded after loading the project `newwidgets.p`. The project includes:

```
prompt.gs
indic.gs
```

4.1 The Prompt Widget

The combination of an entry field and a label is typical for many input dialogues (see Fig. 4.1). Therefore, we would like to extend the library with the composed widget `Prompt`.

Figure 4.1: *Three input masks*

4.1.1 Creating the Prompt Widget

First, we define a new datatype for inputfields. The components of the prompt widget are represented by a tuple of a label and a widget. The prompt widget is parameterized over the type of its input since its entry component is parameterized. The type `Prompt` is a synonym for the window item `W Prompt0`. Additionally we define two selection functions for the new widget.

```
data Prompt0 v = Prompt (Entry v, Label)
type Prompt  v = W (Prompt0 v)
```

```
promptE :: Prompt v -> Entry v
promptE p = let Prompt (e,l) = getWidget p in e

promptL :: Prompt v -> Label
promptL p = let Prompt (e,l) = getWidget p in l
```

The exact representation of window items is irrelevant for us. We use the function `getWidget` to select the widget part of a window item.

Next, we define the construction function for `Prompt`. This function defines the exact layout of the widget.

```
prompt :: HasConfigs (Prompt a) => [Conf (Prompt a)] -> Window -> GUI (Prompt a)
prompt cs w =
  do l <- label [] w
     e <- entry [] w
     composeWidget (Prompt (e,l)) (l << e) cs
```

The function `composeWidget` actually creates and configures the widget. It has the following signature:

```
composeWidget :: (WindowItem (W w), WindowItem (W v), HasConfigs (W w))
                 => w -> W v -> [Conf (W w)] -> GUI (W w)
```

Next, we define `Prompt` as an instance of the class `WindowItem`.

```
instance WindowItem (Prompt v)
```

Now, we have to define `Prompt` as an instance of the desired classes in the user hierarchy (cf. Sect. 2.4).

4.1.2 Configuring New Widgets

How does configuring work for prompt widgets? We have to make sure that the configuration options are correctly distributed over the components of the composed widget. This is done by giving a suitable redefinition of the function `cset`:

```
instance HasConfigs (Entry v) => HasConfigs (Prompt v) where
  cset w c =
    case (c w) of
      (Tk_Text s)      -> cset (promptL w) (const (Tk_Text s))
      (Tk_InitValue x) -> cset (promptE w) (const (Tk_InitValue x))
      otherwise        -> do cset (promptE w) (const (c w))
                             cset (promptL w) (const (c w))
  onArgs e s a = onArgs e s a . promptE
```

We apply the configuration function `c` to the widget `w`, so we get the actual constructor for the configuration option. Using pattern matching, we may now decide whether to apply an option on the label part, the entry part, or on both parts.

In our application, the configuration option `Text` now acts on the label part, `InitValue` on the entry part and all other options on both parts of the prompt widget. Of course, we have to redefine `cget` in a similar way, if we want to read configuration options as well.

Finally, we define `Prompt` as an instance of `HasText` and `HasInput`:

```
instance HasText (Prompt v)
instance HasInput W Entry0 v => HasInput W Prompt0 v where
  getValue = getValue . promptE
  setValue = setValue . promptE
```

All the functionality offered for labels and entries, is automatically available for prompt widgets as well. An application of the prompt widget is the simple adder.

```
ex_prompt :: IO ()
ex_prompt = start $
  do w <- window [ title "Simple Adder 2" ]
     i <- prompt [ text "Press the return key"
                 , initValue 0, self $ on return . updValue (+1) ] w
     pack i
```

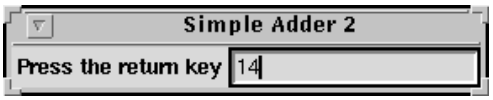

Figure 4.2: *The prompt widget*

In the `tk.prelude` an implementation of the prompt widget exists under the name `input`.

4.2 The Indicator Widget

This example shows a more complicated example of a composed widget. We want to develop a indicator widget, i.e., a widget, displaying a bar that informs us about some percentage (see Fig. 4.3).

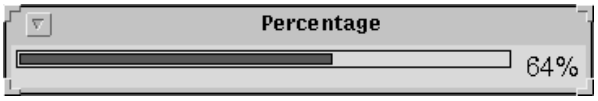

Figure 4.3: *The indicator widget*

The indicator widget has four components, two rectangle widgets that inform us about the status of the indicator, a canvas widget to contain the rectangles and a label to display the percentage. Furthermore, we use a mutable variable to represent the state of the widget. The state is determined by the position of the indicator. This position depends on the actual size of the widget. Therefore, also the actual size is a component of the widget state.

The corresponding datatypes and selection functions are:

```
type IndState = Var (Int,Int,Int)               -- value, width, height
type IndGUI   = (Canvas, Label, CRect, CRect)   -- canvas, border, indicator

data Indicator0 a = Indicator IndGUI IndState a
type Indicator    = W (Indicator0 Int)

canI :: Indicator -> Canvas
canI i = let (Indicator (c,_,_,_) _ _) = getWidget i in c

labI :: Indicator -> Label
```

```
labI i = let (Indicator (_,l,_,_) _ _) = getWidget i in l

borI :: Indicator -> CRect
borI i = let (Indicator (_,_,b,_) _ _) = getWidget i in b

recI :: Indicator -> CRect
recI i = let (Indicator (_,_,_,r) _ _) = getWidget i in r

stateI :: Indicator -> IndState
stateI i = let (Indicator _ st _) = getWidget i in st
```

The type `Indicator0` is parameterized over the type of it contents so we can define it as an instance of the class `HasInput`. Since indicator values are always integers, we instantiate this type variable with `Int`.

The construction function `indicator` first generates a canvas, a label and two rectangles. The actual size of the rectangles is specified by the width and height configuration options.

```
indicator :: [Conf Indicator] -> Window -> GUI (Indicator)
indicator cs w =
  let defaults  = [height 20, width 100, foreground "red"]
  in
  do c <- canvas [] w
     l <- label [width 4, text "0%"] w
     i <- crect (0,0) (0,0) [] c
     j <- crect (0,0) (0,0) [] c
     st <- newState (0,20,100)
     composeWidget (Indicator (c,l,i,j) st 0) (c<|<l) (defaults ++ cs)
```

The next step is to define `Indicator` as an instance of the class `WindowItem` and `HasConfigs`. The function `cset` distributes the configuration options over the several components of the widget. If we want to modify the width or height of the widget, we have to update the widget state.

```
instance WindowItem Indicator

instance HasConfigs Indicator where
  cset w c =
    case (c w) of
        Tk_Height h     -> newheight w h
        Tk_Width h      -> newwidth w h
        Tk_Foreground r -> cset (recI w) (fillColor r)
        Tk_Background r -> do cset (labI w) (background r)
                              cset (canI w) (background r)
                              cset (canI w) (highlightBackground r)
        Tk_Font f       -> cset (labI w) (font f)
        otherwise       -> cset (canI w) (const (c w))
    where
      newheight ind v =
        do (i,x,y) <- readState (stateI ind)
           writeState (stateI ind) (i,x,v)
           let ratio = (fromInteger x / 100.0) * fromInteger i
           setCoords (borI ind) [(3,3),(x+7,v+3)]
           setCoords (recI ind) [(5,5),(5+truncate ratio,5+(v-4))]
           cset (canI ind) (height (v+4))

      newwidth ind v =
        do (i,x,y) <- readState  (stateI ind)
           writeState (stateI ind) (i,v,y)
           let ratio = (fromInteger v / 100.0) * fromInteger i
```

```
setCoords (borI ind) [(3,3), (v+7,y+3)]
setCoords (recI ind) [(5,5), (5+truncate ratio,5+(y-4))]
cset (canI ind) (width (v+8))
```

The function `background` changes the background of both widgets. By changing the `highlightBackground` as well, the widgets really look like 'one' widget.

In `HasConfigs` we defined how to handle configuration options. We still have to define `Indicator` as an instance of the classes that define the desired options:

```
instance HasBackground Indicator
instance HasForeground Indicator
instance HasBorder     Indicator
instance HasWidth      Indicator
instance HasHeight     Indicator
```

Finally, we define `Indicator` as an instance of the class `HasInput`. `getValue` reads the value directly from the widget state. `setValue` writes the new value to the widget state and changes the layout of the indicator-rectangle.

```
instance HasInput W Indicator0 Int where
  getValue w = do (i,_,_) <- readState (stateI w)
                  result i

  setValue w i =
    do (v,x,y) <- readState (stateI w)
       writeState (stateI w) (i,x,y)
       let newx = truncate ((fromInteger x / 100.0) * fromInteger i)
       setCoords (recI w) [(5,5), (5+newx,5+(y-4))]
       cset (labI w) (text (show i ++ "%"))
```

The following example shows an application of the indicator widget. A scaler widget is used to control the indicator. If we move the scaler, the indicator changes correspondingly.

```
ex_indic :: IO ()
ex_indic = start $
  do w <- window []
     i <- indicator [height 10, width 200, background "white"] w
     e <- hscale [scaleRange (0,100), height 200] w
     let cmd = do x <- getValue e; setValue i x
     cset e (command cmd)
     pack (i ^^ e)
```

Chapter 5 Signatures of the `tk.prelude`

This chapter lists the signatures of the user functions of the `tk.prelude`. For the exact implementation of these functions we refer to [VSS96b].

5.1 Start and Quit

```
start :: GUI () -> IO ()
quit  :: GUI ()
```

5.2 User Classes and Instances

```
class Widget w => HasConfigs w where
  cset   :: w -> Conf w -> GUI ()
  cget   :: GUIValue v => w -> (v -> Conf w) -> GUI v
  csets  ::  w -> [Conf w] -> GUI ()
  on     :: TkEvent -> GUI () -> Conf w
  onXY   :: TkEvent -> ((Int,Int) -> GUI ()) -> Conf w  -- relative to screen
  onxy   :: TkEvent -> ((Int,Int) -> GUI ()) -> Conf w  -- relative to widget
  onArgs :: TkEvent -> String -> ([String] -> GUI ()) -> Conf w
            -- see tk-substitution patterns for valid strings

instance Widget a             => HasConfigs a
instance TopLevel (T a)       => HasConfigs (T a)
instance Widget (W a)         => HasConfigs (W a)
instance MenuItem (M a)       => HasConfigs (M a)
instance CanvasItem (C a)     => HasConfigs (C a)
instance HasConfigs (Entry a) => HasConfigs (Input a)
instance HasConfigs Tag
```

```
class HasConfigs w => HasBackground w where
  background :: String -> Conf w
                -- see local rgb.txt file for valid color names

instance HasBackground Window
```

```
instance HasBackground Menu
instance HasBackground Frame
instance HasBackground Scrollbar
instance HasBackground Canvas
instance HasBackground CText
instance HasForeground a => HasBackground a
```

```
class HasBackground w => HasForeground w where
  foreground :: String -> Conf w
  font       :: String -> Conf w
                -- execute `xlsfonts´ for list of valid fonts

instance HasForeground Menu
instance HasForeground (Entry a)
instance HasForeground Edit
instance HasForeground (Listbox a)
instance HasForeground CText
instance HasForeground CBitmap
instance HasText a => HasForeground a
```

```
class HasBackground w => HasBorder w where
  borderWidth :: Int -> Conf w
  cursor      :: String -> Conf w -- see local cursorfont.h
  relief      :: String -> Conf w
              -- valid options are sunken, ridge, flat, raised or groove

instance HasBorder Menu
instance HasWidth a => HasBorder a
```

```
class HasBorder w => HasWidth w where
  width               :: Int -> Conf w
  highlightBackground :: String -> Conf w
  highlightColor      :: String -> Conf w
  highlightThickness  :: Int -> Conf w
  focus               :: w -> GUI ()
  takeFocus           :: Bool -> Conf w

instance HasWidth Scrollbar
instance HasWidth Message
instance HasWidth (Entry a)
instance HasWidth (Entry a) => HasWidth (Input a)
instance HasHeight a => HasWidth a
```

```
class HasWidth w => HasHeight w where
  height :: Int -> Conf w

instance HasHeight Window
instance HasHeight Frame
instance HasHeight Label
instance HasHeight Canvas
instance HasHeight Scale
```

```
instance HasHeight Edit
instance HasHeight (Listbox a)
instance HasHeight Button
instance HasHeight Radiobutton
instance HasHeight Menubutton
instance HasHeight Checkbutton
instance HasHeight (Input a)
```

```
class HasWidth w => HasPad w where
  padx :: Int -> Conf w
  pady :: Int -> Conf w

instance HasPad Label
instance HasPad Message
instance HasPad Edit
instance HasPad Button
instance HasPad Radiobutton
instance HasPad Menubutton
instance HasPad Checkbutton
```

```
class HasForeground w => HasAnchor w where
  anchor  :: String -> Conf w -- n, ne, se, s, sw, w, nw, center
  justify :: String -> Conf w -- left, right, center

instance HasAnchor Label
instance HasAnchor Message
instance HasAnchor Button
instance HasAnchor Radiobutton
instance HasAnchor Menubutton
instance HasAnchor Checkbutton
instance HasAnchor CText
instance HasAnchor CBitmap
```

```
class HasCommand w => HasIndicator w where
  indicatorColor :: String -> Conf w
  indicatorOn    :: Bool -> Conf w

instance HasIndicator Radiobutton
instance HasIndicator MRadiobutton
instance HasIndicator Checkbutton
instance HasIndicator CheckbuttonM
```

```
class HasCoords a => HasCoords a where
  moveObject   :: a -> (Int, Int) -> GUI ()
  removeObject :: a -> GUI ()
  lowerObject  :: a -> GUI ()
  raiseObject  :: a -> GUI ()
  getCoords    :: a -> GUI [(Int,Int)]
  setCoords    :: a -> [(Int,Int)] -> GUI ()

instance CanvasItem (C a) => HasCoords (C a)
```

```
class HasCoords a => HasFillColor a where
  penWidth  :: Int -> Conf a
  penColor  :: String -> Conf a
  fillColor :: String -> Conf a

instance HasFillColor COval
instance HasFillColor CLine
instance HasFillColor CRectangl
```

```
class HasBorder w => HasScroll w

instance HasScroll Canvas
instance HasScroll (Entry a)
instance HasScroll Edit
instance HasScroll (Listbox a)
```

```
class (HasConfigs (c (w v)), GUIValue v) => HasInput c w v where
  getValue  :: c (w v) -> GUI v
  setValue  :: c (w v) -> v -> GUI ()
  updValue  :: c (w v) -> (v -> v) -> GUI ()
  initValue :: v -> Conf (c (w v))
  readOnly  :: Bool -> Conf (c (w v))

instance HasInput T Radio0 Int
instance HasInput W Scale0 a
instance HasInput W Entry0 a
instance HasInput W (Edit0 (Int,Int)) [Char]
instance HasPosition Listbox0 Int [a] => HasInput W (Listbox0 Int) [a]
instance Widget (a (Checkbutton0 b)) => HasInput a Checkbutton0 b
instance HasInput W Entry0 a => HasInput W Input0 a
```

```
class (HasInput W (w p) v, GUIValue p, Position p)
  => HasPosition w p v where
  putBegin          :: (W (w p v)) -> v -> GUI ()
  putEnd            :: (W (w p v)) -> v -> GUI ()
  putPos            :: (W (w p v)) -> p -> v -> GUI ()
  getFromTo         :: (W (w p v)) -> p -> p -> GUI v
  getSize           :: (W (w p v)) -> GUI p
  delFromTo         :: (W (w p v)) -> p -> p -> GUI ()
  setYView          :: (W (w p v)) -> Int -> GUI ()
  getSelection      :: (W (w p v)) -> GUI [p]
  setSelection      :: (W (w p v)) -> [p] -> GUI ()
  selectBackground  :: String -> Conf (W (w p v))
  selectForeground  :: String -> Conf (W (w p v))
  selectBorderWidth :: Int -> Conf (W (w p v))

instance HasPosition Edit0 (Int,Int) [Char]
instance HasPosition Listbox0 Int [a]
```

```
class HasForeground w => HasText w where
```

```
  text      :: String -> Conf w
  bitmap    :: String -> Conf w  -- bitmap file name
  underline :: Int -> Conf w

instance HasText Label
instance HasText Message
instance HasText Scale
instance HasText Tag
instance HasText CText
instance HasText CBitmap
instance HasText (Input a)
instance HasCommand a => HasText a
```

```
class HasText w => HasCommand w where
  command          :: GUI () -> Conf w
  active           :: Bool -> Conf w
  activeBackground :: String -> Conf w
  activeForeground :: String -> Conf w
  invoke           :: w -> GUI ()

instance HasCommand Scale
instance HasCommand Button
instance HasCommand ButtonM
instance HasCommand Menubutton
instance HasCommand MenubuttonM
instance HasIndicator a => HasCommand a
```

```
data OkOrErr a = Tk_Ok a | Tk_Err String

class (Text g) => GUIValue g where
  tk_convert         :: String -> OkOrErr g
  tk_defaultValue    :: g
  tk_toGUI           :: g -> String
  tk_fromGUI         :: String -> GUI g
```

5.3 TopLevel items

```
type Window = T Window0
  window         :: [Conf Window] -> GUI Window
  windowDefault  :: [Conf Window] -> [Conf Default] -> GUI Window
  closeWindow    :: Window -> GUI ()
  openWindow     :: [Conf Window] -> (Window -> GUI (W w)) -> GUI ()
  openDefault    :: [Conf Window] -> [Conf Default] ->
                    (Window -> GUI (W w)) -> GUI ()
  pack           :: W a -> GUI ()
  packDefault    :: W a -> [Conf Default] -> GUI ()
  title          :: String -> Conf Window
  winSize        :: (Int,Int) -> Conf Window
  winPosition    :: (Int,Int) -> Conf Window

type Menu = T Menu0
  menu           :: HasConfigs (c Menubutton0)
```

```
                => [Conf Menu] -> c Menubutton0 -> GUI Menu
  menuDefault   :: HasConfigs (c Menubutton0)
                => [Conf Menu] -> [Conf Default] -> c Menubutton0 -> GUI Menu
  popup         :: (Int, Int) -> Menu -> GUI ()

type Radio = T (Radio0 Int)
  radio         :: (HasConfigs (c Radiobutton0))
                => [Conf Radio] -> [c Radiobutton0] -> GUI Radio
```

5.4 Window Items

```
type Frame = W Frame0
  frame         :: WindowItem (W a) => W a -> [Conf Frame] -> GUI Frame

type Scrollbar = W Scrollbar0
  scrollbar     :: [Conf Scrollbar] -> Window -> GUI Scrollbar
  hscroll       :: HasScroll (W w)
                => [Conf Scrollbar] -> W w -> GUI Scrollbar
  vscroll       :: HasScroll (W w)
                => [Conf Scrollbar] -> W w -> GUI Scrollbar

type Label = W Label0
  label         :: [Conf Label] -> Window -> GUI Label

type Message = W Message0
  message       :: [Conf Message] -> Window -> GUI Message
  aspect        :: Int -> Conf Message

type Canvas = W Canvas0
  canvas        :: [Conf Canvas] -> Window -> GUI Canvas
  scrollRegion  :: (Int,Int) -> Conf Canvas
  clearCanvas   :: Canvas -> GUI ()

type Scale = W (Scale0 Int)
  vscale        :: [Conf Scale] -> Window -> GUI Scale
  hscale        :: [Conf Scale] -> Window -> GUI Scale
  scaleRange    :: (Int,Int) -> Conf Scale
  sliderLength  :: Int    -> Conf Scale
  tickInterval  :: Int    -> Conf Scale
  troughColor   :: String -> Conf Scale

type Entry a = W (Entry0 a)
  entry         :: HasConfigs (Entry a)
                => [Conf (Entry a)] -> Window -> GUI (Entry a)

type Edit = W (Edit0 (Int,Int) [Char])
  edit          :: [Conf Edit] -> Window -> GUI Edit
  wrap          :: Bool -> Conf Edit

  data Mark = Mark String
    setMark       :: Edit -> (Int,Int) -> GUI Mark
    getMark       :: Edit -> Mark -> GUI (Int,Int)

  type Tag = Tag0 ()
    tag           :: [(Int,Int)] -> [Conf Tag] -> Edit -> GUI Tag
    putPosTag     :: Edit -> (Int,Int) -> String -> [Conf Tag] -> GUI Tag
    putEndTag     :: Edit -> String -> [Conf Tag] -> GUI Tag
    delTag        :: Tag -> GUI ()
    tagRange      :: Tag -> GUI [(Int,Int)]
    lowerTag      :: Tag -> GUI ()
```

```
type Listbox a = W (Listbox0 Int a)
  listbox        :: HasConfigs (Listbox a)
                 => [Conf (Listbox a)] -> Window -> GUI (Listbox a)
  multipleSelect :: Bool -> Conf (Listbox a)


type Button  = W Button0
  button         :: [Conf Button] -> Window -> GUI Button


type Radiobutton  = W Radiobutton0
  radiobutton    :: [Conf Radiobutton] -> Window -> GUI Radiobutton


type Menubutton = W Menubutton0
  menubutton     :: [Conf Menubutton] -> Window -> GUI Menubutton


type Checkbutton  = W Checkbutton0
  checkbutton    :: [Conf Checkbutton] -> Window -> GUI Checkbutton
```

5.5 Menu Items

```
type MButton = M Button0
  mbutton        :: [Conf MButton] -> Menu -> GUI MButton

type MRadiobutton = M Radiobutton0
  mradiobutton   :: [Conf MRadiobutton] -> Menu -> GUI MRadiobutton

type Cascade    = M Menubutton0
  cascade        :: [Conf Cascade] -> Menu -> GUI Cascade

type MCheckbutton = M Checkbutton0
  mcheckbutton   :: [Conf MCheckbutton] -> Menu -> GUI MCheckbutton

type Separator = M Separator0
  separator      :: Menu -> GUI Separator
```

5.6 Canvas Items

```
type COval = C Oval0
  coval  :: (Int,Int) -> (Int,Int) -> [Conf COval] -> Canvas -> GUI COval

type CLine = C Line0
  cline  :: (Int,Int) -> (Int,Int) -> [Conf CLine] -> Canvas -> GUI CLine

type CRect = C Rect0
  crect :: (Int,Int) -> (Int,Int) ->
           [Conf CRectangle] -> Canvas -> GUI CRectangle

type CText = C CText0
  ctext :: (Int,Int) -> [Conf CText] -> Canvas -> GUI CText

type CBitmap = C CBitmap0
  cbitmap :: (Int,Int) -> [Conf CBitmap] -> Canvas -> GUI CBitmap
```

5.7 User Defined Events

```
key                                          :: String -> TkEvent
click, doubleClick, motion                   :: Int -> TkEvent
return                                       :: TkEvent
cursorUp, cursorDown, cursorLeft, cursorRight :: TkEvent
```

5.8 Widget Combinators and Layout Functions

```
infixl 7 <<, <*<, <-<, <|<, <+<, <*-<, <*|<, <*+<
infixl 6 ^^, ^*^, ^-^, ^|^, ^+^, ^*-^, ^*|^, ^*+^

(<<),(<*<), (<-<),(<|<), (<+<),(<*-<), (<*|<),(<*+<)
  :: (WindowItem (W a),WindowItem (W b)) => W a -> W b -> Frame
(^*^),(^^),(^-^),(^|^),(^+^),(^*-^),(^*|^),(^*+^)
  :: (WindowItem (W a),WindowItem (W b)) => W a -> W b -> Frame

matrix               :: WindowItem (W a) => Int -> [W a] -> Frame
horizontal, vertical :: WindowItem (W a) => [W a] -> Frame
fillX, fillY, fillXY :: WindowItem (W a) => W a -> Frame
expand,flexible      :: WindowItem (W a) => W a -> Frame
```

5.9 Monads and Variables

```
infixr 1 ==>
(==>)      :: GUI Bool -> GUI () -> GUI ()
doneM      :: Monad m => m ()
seq        :: Monad m => m a -> m b -> m b
void       :: GUI a -> GUI ()
seqs       :: Monad m => [m ()] -> m ()
binds      :: Monad m => [m a] -> m [a]

newState   :: a -> GUI (Var a)
readState  :: Var a -> GUI a
writeState :: Var a -> a -> GUI ()
modState   :: Var a -> (a -> a) -> GUI ()
```

5.10 Miscellaneous

```
self       :: (a -> a -> b) -> a -> b
rgb        :: Int -> Int -> Int -> String
numval     :: String -> Int
startClock :: Int -> GUI () -> GUI ClockId
stopClock  :: ClockId -> GUI ()
updateTask :: GUI ()
```

5.11 Composing Widgets

```
composeWidget :: (WindowItem (W w), WindowItem (W v), HasConfigs (W w))
              => w -> W v -> [Conf (W w)] -> GUI (W w)
```

```
input          :: HasConfigs (Input v)
               => [Conf (Input v)] -> Window -> GUI (Input v)
inputE         :: Input v -> Entry v
inputL         :: Input v -> Label
```

Bibliography

[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.

[BR89] P. Briggs and J. Ramsdell. *NuwWeb Version 0.87b: A Simple Literate Programming Tool*, 1989. available by ftp from `ftp.dante.de`.

[BW89] R. Bird and Ph. Wadler. *Introduction to Functional Programming*. Prentice Hall International, 1989.

[CH93] M. Carlsson and Th. Hallgren. Fudgets — graphical user interfaces and I/O in lazy functional languages. Licentiate Thesis, May 1993.

[HS95] M. Hermenegildo and S.D. Swierstra, editors. *Programming Languages: Implementations, Logics and Programs. 7th International Symposium, PLILP '95*, volume 982 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1995.

[JD93] M.P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993.

[JM95] J. Jeuring and E. Meijer, editors. *Advanced Functional Programming. First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 1995*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Jon93a] M.P. Jones. *An introduction to Gofer (draft)*, 1993. Included as part of the standard Gofer distribution.

[Jon93b] M.P. Jones. *Release notes for Gofer 2.28*, 1993. Included as part of the standard Gofer distribution.

[Jon95] M.P. Jones. Functional programming with overloading and higher-order polymorphism. In Jeuring and Meijer [JM95], pages 97–136.

[LPJ94] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. Technical report, University of Glasgow, November 1994.

[LPJ95] J. Launchbury and S.L Peyton Jones. State in haskell. *Lisp and Symbolic Computation*, (8):293–341, 1995.

[NR95] R. Noble and C. Runciman. Gadgets: Lazy functional components for graphical user interfaces. In Hermenegildo and Swierstra [HS95], pages 321–340.

[Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, 1994.

[PJW93] 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.

[Sch96] T. Schwinn. Funktionale implementierung grafischer benutzeroberflächen. Master's thesis, Universität Ulm, Fakultät für Informatik, 1996. in German.

[VSS96a] T. Vullinghs, W. Schulte, and T. Schwinn. The design of a functional gui library using constructor classes. In *Perspectives of System Informatics*, Novosibirsk, 1996.

[VSS96b] T. Vullinghs, T. Schwinn, and W. Schulte. Tkgofer: Implementation notes and reference manual. Technical report, Universität Ulm, Fakultät für Informatik, 1996. to appear.

[VTS95] T. Vullinghs, D. Tuijnman, and W. Schulte. Lightweight GUIs for functional programming. In Hermenegildo and Swierstra [HS95], pages 341–356.

[Wad90] Ph. Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[Wad95] Ph. Wadler. Monads for functional programming. In Jeuring and Meijer [JM95], pages 24–52.

Index