Embracing Windows

Colin J. Taylor

Department of Computer Science, University of Nottingham,
University Park, Nottingham, NG7 2RD, England.
cjt@cs.nott.ac.uk

Technical Report NOTTCS-TR-96-1

October 23, 1996

Abstract

There are a number of systems that advocate the use of lazy functional languages for the programming of graphical user interfaces (GUIs) such as Haggis, Gadgets, and Fudgets. These systems have addressed similar issues, such as how to handle I/O in a purely functional language, and how to provide a structured interface to the event driven model of windowing systems. In this report we present a framework that encapsulates such common elements and is intended to aid in the formal study of the relationships between these systems. The framework is called Embracing Windows, partly because it embraces the window paradigm, and partly because it uses the Hugs functional programming system.

Two high-level GUI development systems have been built on the framework, and are based upon Haggis and Fudgets, respectively. The essence of these systems is distilled and a brief comparison made. The complete Haskell source code for the Embracing Windows framework is presented as an appendix. The framework also illustrates the usefulness of Haskell type classes for structuring GUI development systems.

Keywords: Graphical User Interfaces; Monads; Fudgets; Haggis; Type Classes; Functional Programming.

Contents

1 Introduction **5**
1.1 I/O in Purely Functional Languages 5
1.2 Monads 6

2 Interfacing to a Window System **9**
2.1 Modern Window Systems 9
2.2 Monadic Primitives 9
2.2.1 Types 11
2.3 Event Loops 12
2.4 Message Cracking 13
2.5 Building Event Handlers 14
2.5.1 The GUI Monad 14
2.5.2 The GUI State 16
2.5.3 A Simple Event Handler 17
2.5.4 A Better Interface 17
2.6 Drawing Graphics 18
2.6.1 Graphic Primitives 18
2.6.2 The Draw Monad 18
2.7 Controls 20
2.7.1 Text Labels 21
2.7.2 Edit Fields 21
2.7.3 Buttons 22
2.8 The Counter Program 23
2.9 Implementation Details 24
2.9.1 Events 24
2.9.2 Controls 24

3 Widgets **26**
3.1 The Representation of Widgets 26
3.2 A Library of Widgets 27
3.3 Composite Widgets 27
3.4 Layout of Widgets 27
3.4.1 Implementation of Layout 29
3.5 Stateful Widgets 29
3.6 Comparison to Haggis 30

4 **Fudgets** 33
4.1 Overview of Fudgets . . . 33
4.2 Implementation of Fudgets . . . 35
4.3 Layout of Fudgets . . . 36
4.4 Looping Combinators . . . 38

5 **The Essence of Functional GUI's** 41

6 **Conclusions and Future Work** 43
6.1 Acknowledgements . . . 45

A **Example Applications** 46
A.1 A Calculator . . . 46
A.1.1 The Calculator State Machine . . . 46
A.1.2 A Widget Graphical User Interface . . . 48
A.1.3 A Fudget Graphical User Interface . . . 50
A.1.4 Comparison of Widget and Fudget GUIs . . . 52
A.2 A Combination Lock . . . 52
A.2.1 A Widget Combination Lock . . . 52

B **Source Code** 55
B.1 I/O Primitives and Library Files . . . 55
B.1.1 Types . . . 55
B.1.2 Monadic Primitives . . . 56
B.1.3 Window Constants . . . 57
B.1.4 Table . . . 58
B.1.5 Event loop primitives . . . 58
B.1.6 Window system primitives . . . 59
B.1.7 Mutable Variables . . . 60
B.2 Event Handling . . . 61
B.2.1 GUI . . . 61
B.2.2 Event Handlers . . . 62
B.2.3 Message . . . 64
B.2.4 Lifted Functions . . . 65
B.3 The Core . . . 66
B.3.1 Windows . . . 66
B.3.2 On Handlers . . . 68
B.3.3 Controls . . . 69
B.3.4 Graphics . . . 71
B.3.5 Embracing Windows Framework . . . 74
B.4 Widgets . . . 75
B.4.1 Widgets . . . 75
B.4.2 Layout Widgets . . . 77
B.4.3 Standard Widgets . . . 79
B.4.4 Widgets System . . . 81
B.5 Fudgets . . . 82
B.5.1 Fudgets . . . 82
B.5.2 Layout Fudgets . . . 83

B.5.3 Standard Fudgets 87
B.5.4 Stream processors 88
B.5.5 Stream Processor Combinators 89
B.5.6 Stream Processor Operations 89
B.5.7 Fudget Combinators 90
B.5.8 Fudgets System 91

Chapter 1

Introduction

There are a number of systems for describing the development of graphical user interfaces (GUIs) in a lazy functional language; examples include Fudgets [9], Gadgets [16], and Haggis [6]. All of the designers of these systems have had to grapple with similar issues, such as how to perform side effecting I/O in a purely functional language, and how to provide a structured interface to the event driven model of windowing systems. However, the main research thrust of these systems is in the abstractions they provide for the construction of GUIs. In this report, we describe a framework that encapsulates the common elements of some of these systems, and consists of layers, each building on top of the functionality provided by the lower layers. Two high-level systems for constructing GUIs have been built on the framework, based on Haggis and Fudgets respectively.

The Embracing Windows[1] framework has been developed using the Hugs functional programming system [10] and currently interfaces to Windows 95[2]. The examples in this report assume familiarity with Haskell 1.3 [17].

In the remainder of this chapter, we discuss the lowest layer in the framework, concerning I/O in a purely functional language. Chapter 2 details the next layer, which encapsulates a basic interface to a window system. In Chapters 3 and 4 we describe the two high-level systems for constructing GUIs that have been built on the framework. Chapter 5 summarises these two systems and briefly describes how they can be used together. Finally we conclude in Chapter 6 with a survey of related work, and also present some ideas for future investigation. The appendices present a number of extended example applications built with the high-level systems described in Chapters 3 and 4. The Haskell source code of the framework is also included in the appendices.

1.1 I/O in Purely Functional Languages

This section discusses I/O in purely functional languages, which form a basis for the particular form of I/O required by GUIs. A first attempt at I/O in a functional language might be to provide side effecting primitives similar to I/O functions in imperative languages such as C:

```
getChar :: Char
putChar :: Char -> ()
```

[1]The Embracing Windows framework is available from `http://www.cs.nott.ac.uk/~cjt/EW.html`

[2]Windows 95 is a registered trademark of the Microsoft Corporation

The `getChar` function waits for a key to be pressed and returns the corresponding character, while `putChar c` prints the character given as its argument, `c`, on the display. However, primitives like these limit the ability to use simple equational reasoning and program transformation due to the loss of referential transparency. The result of a referentially transparent function should depend purely upon its arguments. Every time `getChar` is called it could potentially return a different value, thus destroying referential transparency, for example:

```
x == x where x = getChar  ≠  getChar == getChar
```

These expressions are not equivalent because the result of the side effecting primitives is shared, rather than sharing the actions that the side effecting primitives perform.

A number of different approaches to handling I/O in functional languages that solve these problems have been proposed:

- **Streams**. A stream is a lazy list of data objects. Miranda[3] [24], Ponder [4] and Hope [2] use streams for I/O. In these systems, an interactive program is modelled as a function from one stream representing the input, to another representing the output.
- **Continuations**. Continuation passing style entails writing functions that take an extra argument, a continuation, describing what to do next. Instead of the function returning its result directly, it is passed on to the continuation to be processed first. A useful property of continuations is their ability to specify an order of evaluation [18]. This can form the basis for a model of I/O using side effecting primitives.
- **Monads**. Motivated by the work of Moggi [13] and Spivey [23], Wadler [27, 26] proposed a style of functional programming based on monads that can be used to model impure 'features' such as input and output. Monadic I/O [11] uses a collection of combinators to build interactive programs from primitive actions. When the program is executed, these actions are performed, realizing the I/O. Monadic I/O is used as the mechanism for I/O in Haskell 1.3 [17].

1.2 Monads

The use of monads is now established as a method for describing interactive programs, and will be used as the mechanism for I/O throughout this report. Our original example can be described using the `IO` monad as:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

The result of the `putChar` function is an action that prints a character on the display. Similarly, the result of `getChar` is an action that reads a character from the keyboard. Monadic combinators are used to combine actions together:

```
returnIO :: a -> IO a
thenIO   :: IO a -> (a -> IO b) -> IO b
```

[3] Miranda is a trademark of Research Software Ltd.

The `returnIO` combinator constructs a trivial action that has no side effects, and whose return value is the first argument of the combinator. `a 'thenIO' f`, when performed, will first perform the action `a`, applying the result to the function f to obtain a further action. This latter action will be performed next and the value returned by this action is the result of the expression `a 'thenIO' f`. A monad is simply an abstract data type, specified by a type constructor, `m`, which supports two operations whose types are the same as the types of the functions `returnIO` and `thenIO` with the type constructor `IO` replaced by `m`. A monad may also support other operations as well. Three algebraic properties must be satisfied by the two operations, but we do not concern ourselves further with these properties in this report. Together the data type `IO` and the monadic combinators `returnIO` and `thenIO` form a monad.

The type constructor `IO` can be thought of as being defined as:

```
type IO a = World -> (a, World)
```

Here, a value of the type `World` represents the state of the 'real world', such as the contents of a file system, pictures displayed on a screen, and characters read from a keyboard. This definition is not visible to the programmer and is built into the system as a primitive type. However, it allows us to see why monads restore the referential transparency that our side-effecting primitives lacked. Expanding out the definition of `IO Char` in the type of the `getChar` function gives:

```
getChar :: World -> (Char, World)
```

The return value of this function contains the character pressed and the new state of the world. It is also clear that the character pressed depends upon the initial state of the world, which is the first argument to the function. The result of the function thus depends purely upon its argument, and so maintains referential transparency. We are also free to use equational reasoning, as an expression of type `IO a` represents an action rather than the result of a side-effecting operation. An action can be shared like any other first class value, and is only turned into a side-effecting operation when the program is executed.

The abstract data type that forms the `IO` monad essentially prevents the world from being duplicated. This allows us to optimise operations such as `getChar` to update the value of the world in place, rather than creating a copy of the initial value of the world, modified according to the operation being performed. This results in an efficient implementation of `IO` that still maintains referential transparency.

Using these combinators, we can describe a function that will, print a string on the display:

```
putString        :: String -> IO ()
putString []      = returnIO ()
putString (c:cs) = putChar c 'seqIO'
                   putString cs
```

The `seqIO` combinator is similar to `thenIO`, except that it discards the result of the action specified by its first argument, and only requires an action rather than a function as its second argument. The `seqIO` combinator can be defined in terms of `thenIO` as:

```
seqIO :: IO a -> IO b -> IO b
a 'seqIO' b = a 'thenIO' (\_ -> b)
```

The two combinators, returnIO and thenIO are specific cases of combinators that are required for a data type to be a monad. The standard prelude of Haskell 1.3 includes a type class specifically for representing monads called Monad. This type class supports three overloaded functions, return, >>=, and >> corresponding to returnIO, thenIO and seqIO, respectively. Haskell 1.3 also provides a syntactic convention for expressing functions using monadic combinators, called *do notation*. Using this syntax, the putString function previously examined can be rewritten as:

```
putString       :: String -> IO ()
putString []     = return ()
putString (c:cs) = do putChar c
                      putString cs
```

The remainder of this report will make use of this syntactic convention, which expresses monadic functions more concisely.

Chapter 2

Interfacing to a Window System

In this chapter, we present the core of the Embracing Windows system. The structure of the core is split into five main layers that can be seen in Figure 2.1. We first present an introduction to the concepts involved in modern windowing systems, followed by a description of each of the five main pieces comprising the core.

2.1 Modern Window Systems

Most modern windowing systems allow the user to run multiple applications, with the user switching between applications at will. Such systems are driven by the order of events because the input to applications is essentially interleaved. An event can correspond to external stimuli, such as mouse clicks, or keyboard input, but can also signal an internal operation such as the creation or destruction of a window. The dispatching of events to the appropriate application is handled by the windowing system. Conventionally, applications that are to be used in such a system are built around an *event loop* that processes events. An event loop can be modelled functionally as a stream processor. Such a stream processor transforms a list of events into a list of actions by mapping an appropriate event processing function over the list of events.

Windowing systems encapsulate the environment for drawing graphics in a special structure, commonly referred to as a graphics or device context. This includes information such as the current drawing colour, width and shape of pen, and brush type for filling areas. When an application draws in a window, it must first obtain an appropriate device context. All drawing operations require the device context as a parameter. Finally, when the application has finished drawing it must relinquish the device context to the windowing system.

2.2 Monadic Primitives

Non-graphical interactive programs use primitives that write characters to the display and that read characters from the keyboard. A graphical interactive program requires I/O primitives in a similar way to a non-graphical interactive program. However, the primitives will draw lines in windows, and read clicks from mouse buttons. Instead of reading characters directly from the keyboard, a primitive will supply the characters typed when the input focus is on the application's window. Similarly, instead of writing characters to the display, a primitive will display a character in one of the application's windows.

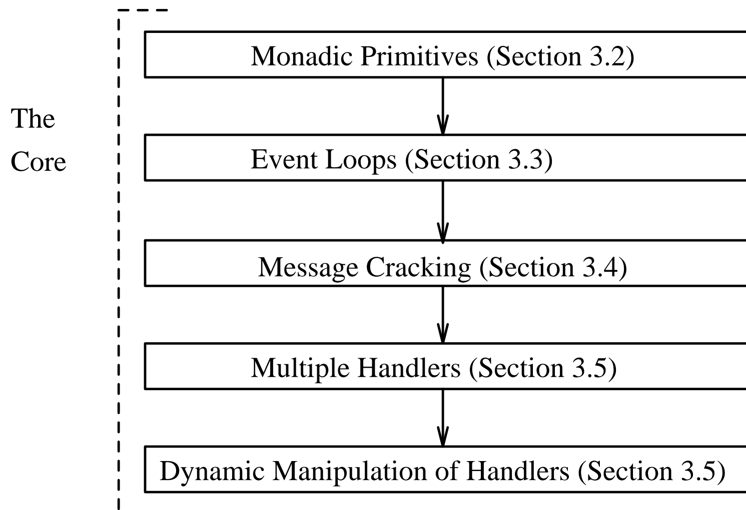


Figure 2.1: The core of the Embracing Windows system

The monadic `putChar` and `getChar` functions provided appropriate primitives for functional programs to interact with a console. In a similar manner, we can define primitives that are appropriate for graphical interactive programs. In the Embracing Windows framework there are three groups of primitives:

- Event loop operations
- Basic window operations, such as creating, and destroying windows.
- Graphic operations, such as drawing lines, and changing pen colours and sizes.

We can use type classes to provide an interface to such primitives (see Figures 2.2 and 2.6). The advantage of using type classes is that we can abstract from the monad that the primitives are defined in. This becomes useful when we need to use the primitives in monads other than the monad that they are originally implemented in. Another advantage of using type classes is that they structure the world effectively. Instead of considering the world as a single entity we can think of it as being comprised of subworlds, such as an event system, a windowing system, and a drawing system, which in turn may consist of more subworlds themselves. This structure is made evident in the type system by the use of type classes. The Clean language structures the world in a similar way [1] and claims that this avoids the specification of ordering between I/O involving separate parts of the world. When two independent subworlds are being manipulated then the order in which they are changed is irrelevant, and the underlying implementation of the language can decide which order to use. The type of a function using the monadic primitives will include a context which expresses precisely the components of the world used, such as event loops, basic window operations, or graphics operations.

The first group of primitives characterises event-driven applications and are defined in the `EventSystem` type class. The second group of primitives characterises applications with window based interfaces and are defined in the `WindowSystem` type class. The last group of primitives characterises applications that draw graphical pictures in windows, and these primitives are described in Section 2.6.1. An instance of both the `EventSystem` and `WindowSystem`

```
class Monad m => EventSystem m where
   eventLoop      :: (Event -> IO Int) -> m ()          -- Starts an event loop
   defaultHandler :: Event -> m Int                     -- A default event handler
   quitEventLoop  :: m ()                               -- Terminate event loop

class EventSystem m => WindowSystem m where
   createShellWindow :: String -> m Window              -- Creates a shell window
   destroyWindow     :: Window -> m ()                  -- Destroys a window
   setWindowCaption  :: Window -> String -> m ()        -- Set window caption
   getWindowCaption  :: Window -> m String              -- Get window caption
   setWindowRect     :: Window -> Rect -> m ()          -- Set window size
   getWindowRect     :: Window -> m Rect                -- Get window size
   getWindows        :: m [Window]                      -- Get open windows
   showWindow        :: Bool -> Window -> m ()          -- Set window visibility
```

Figure 2.2: Window system primitives

type classes is required for the `IO` monad, with the implementations of the methods simply being the monadic primitives built-in to the functional language:

```
instance EventSystem IO where
    ...

instance WindowSystem IO where
    ...
```

The full instance declarations are defined in Appendices B.1.5 and B.1.6. An example of an application that may be event-driven, but not use a window based interface, is a network database server. In a similar way that the window operations of the `WindowSystem` type class are layered on the event-driven operations of the `EventSystem` type class, one could imagine defining a type class encapsulating networking operations.

We can proceed to create monadic functions that give the functional programmer access to all of the functionality of windowing systems that is available to the imperative programmer. However, handling input in windowing systems becomes more involved; modern windowing systems support the use of multiple applications simultaneously, and input to the different applications can be interleaved arbitrarily by the user. We will return to this issue in Section 2.3.

2.2.1 Types

As well as requiring built-in primitives to access the functionality of a window system, built-in types are required. These types are used to represent windows, objects for drawing graphics such as pens and brushes, and device contexts encapsulating the current drawing environment:

```
data Window
data Object
data DC
```

Vectors, represented as pairs of integers are used very frequently when constructing a GUI, to describe positions, sizes, and rectangles. We define type synonyms to make the use of vectors clear:

```
type Vector = (Int, Int)

type Point  = Vector
type Size   = Vector
type Rect   = (Vector, Vector)
```

A disadvantage of using type synonyms is that a value representing a point could be used in the context where a size was expected. Such an inconsistency would not be detected as a type error. This problem could be solved by using datatypes, but would require the insertion of the names of appropriate data type constructors.

Events that occur in a windowing system are represented by values of the type `Event`. The definition of this type is specific to a windowing system (Section 2.9.1 defines an `Event` type for Windows 95). In general, events can be classified by the type of the event, such as mouse button click, key press, or creation of a window. It will be useful to determine the type of an event and, for this purpose, we define a function, `getEventType`, and an associated datatype specifying a domain of event types:

```
data EventType = ...

getEventType :: Event -> EventType
```

The exact implementation of this type and function are dependent upon the windowing system being used. However, it is important that we have a set of values that can be used to represent the types of events.

2.3 Event Loops

Considering an interactive application as a stream processor taking a list of events to an action, and describing this by mapping a function over the list of events leads to a possible implementation:

```
event_loop              :: (Event -> IO ()) -> [Event] -> IO ()
event_loop handle_event = sequence . map handle_event
```

The processing of each individual event results in an action of type `IO ()`, and all of these actions are combined sequentially into one single action using `sequence`, which is a standard Haskell 1.3 function. We can make use of the primitive functions for performing windowing operations, such as drawing lines in windows, in response to events. The `handle_event` function defines the behaviour of the application in response to events.

The `event_Loop` function described above makes the list of events explicit; in practice a list is not used as only one event can happen at a time. The `eventLoop` primitive functions in an imperative manner, waiting for an event to happen and then calling the event handling function specified by its first argument to process this event, and then waiting for the next event, and so on. A list of actions is generated, each one corresponding to the appropriate response for an event, but all these events happen sequentially, and so again an explicit list is not required as the actions can be performed when the events occur. The event handling

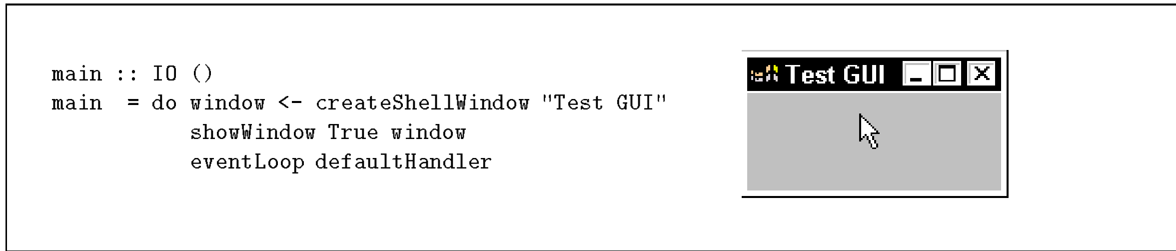


Figure 2.3: A simple graphical application

function can specify default behaviour in response to an event, by passing the event to the windowing system using the default event handler, `defaultHandler`. Finally, a functional program may want to halt the event loop and can do so by using the `quitEventLoop` primitive.

An application with a simple GUI can now be constructed that creates a window and starts an event loop, passing every event to the default event handler (see Figure 2.3). By default the `createShellWindow` function does not make the new window visible, and so the `showWindow` primitive must be used to make it visible.

2.4 Message Cracking

The `Event` datatype is used to encapsulate information about *any* event that may occur in the system. Unfortunately, this means that information pertaining to the event may be packed into the event data structure, and needs to be unpacked before it can be used. The packing algorithm depends on the type of event, and we would like the unpacking to be done automatically for the programmer. The unpacking of this information is commonly referred to as *message cracking*. Ideally, we would like the programmer to be able to write functions that deal with the unpacked data explicitly. For example, the programmer could write a function such as:

```
processLButtonDown :: Vector -> IO ()
```

to specify the behaviour of the application in response to the left mouse button being pressed. The first argument to this function represents the position of the mouse pointer when the left mouse button was pressed. However, this function is not an event handler. To turn it into an event handler, we can use a coercion function such as:

```
handleLButtonDown :: (Vector -> IO ()) -> Event -> IO ()
```

This function manages the unpacking of information from an event, and passes the unpacked information to its first argument to handle the event. Now the programmer can handle a single event in a structured fashion by using a combination of such functions, for example:

```
myEventHandler :: Event -> IO ()
myEventHandler  = handleLButtonDown processLButtonDown
```

The coercion functions are provided as a library for use by the programmer (Appendix B.2.2), allowing the programmer to write functions describing the behaviour of their application without being concerned about the packing of event information into the event data type.

2.5 Building Event Handlers

An application may, in general, be interested in many different types of events, giving different behaviour in each case. Such behaviour could be created by using combinators to combine event processing functions. One of the disadvantages of using combinators is the difficulty of expressing the deletion of a behaviour; the usual solution is to use a combinator to compose a function that overrides the existing behaviour. When specifying the initial behaviour of a GUI combinators suffice, but the behaviour of a GUI may change as the application executes. Combinators do not seem to be particularly natural to express this change in behaviour, as it is not often that the response to an event is incremently changed, instead it is usually just replaced entirely. For this reason, the Embracing Windows framework uses a datatype to represent behaviour, as replacing existing behaviour is simply a matter of replacing one value of the datatype with a different value.

Since events are associated with a particular window, we may also want different behaviour for different windows. We can model this by using a table whose entries specify the behaviour for specific windows. The behaviour of a specific window can also be modelled by a table whose entries determine the responses to specific events. A desirable property is for the behaviour of an application to be able to be altered in response to events.

2.5.1 The GUI Monad

The tables required to specify the behaviour of an application can be implemented functionally. However, this requires that the state be passed explicitly to all functions that may access or modify the state. To avoid having to plumb the state through the system, we can hide the details by using a monad. The monad must be an extension of the `IO` monad so that we can still make use of the monadic primitives from Figure 2.2. Extending the `IO` monad with state transformer capabilities allows us to make the plumbing of the state implicit. However, it was originally thought that using the well known formulation of state transformers, `s -> (s, a)`, would complicate the event loop primitives. An alternative to using the standard formulation of state transformers is to use an `IO` monad with state reader capabilities, where the state is stored in a mutable variable, `Ref s -> IO a`. An interesting question is whether such a monad is equivalent to the conventional state transformer monad; intuitively, at least, it seems that these two approaches should be equivalent.

Mutable variables provide a mechanism to mutate state in such a way that the changes can be implemented by updating the state in-place. A mutable variable has a type `Ref a`, and is a reference to a piece of state containing a value of type `a`. The state contains a mapping from references to values. Operations on mutable variables are allocation, reading, and writing:

```
newRef :: a -> IO (Ref a)
getRef :: Ref a -> IO a
setRef :: Ref a -> a -> IO ()
```

The mutable variables used here are embedded in the `IO` monad, so that they can be used in the context of the primitive window functions such as `createShellWindow`. The function `newRef` takes an initial value and returns an action that creates a new reference, bound to the initial value, and that returns the reference as its result. Given a reference `v`, `getRef v` is an action that performs no side-effects, but uses the state to map the reference to its value. The function `setRef` is an action that modifies the state so that it maps the reference to a

```
class Monad m => MutVars m where
   newRef :: a -> m (Ref a)
   getRef :: Ref a -> m a
   setRef :: Ref a -> a -> m ()
```

Figure 2.4: Mutable variable primitives

new value. It is useful to create a type class encapsulating the operations that characterise mutable variables. This allows the same name to be used for the operations independent of the particular monad they are being used in. The type class can be seen in Figure 2.4. An instance of this type class for the `IO` monad is implemented using the built-in primitives for mutable variables in the Hugs system (the instance declaration is shown in full in Appendix B.1.7).

The Embracing Windows framework captures the behaviour of an application as the state in a state reader monad. This state stores a reference to the tables that specify the responses of the application to particular events:

```
type GUIState = WindowTable
type GUIStateVar = Ref GUIState

newtype GUI a = GUI (GUIStateVar -> IO a)
```

Appropriate `unitGUI` and `bindGUI` functions can be defined, and used to define `GUI` as an instance of the `Monad` type class. Instances of the `EventSystem`, `WindowSystem` and `MutVars` type classes can also be defined by using the `liftGUI` function that lifts an operation from the `IO` monad to the `GUI` monad:

```
liftGUI  :: IO a -> GUI a
liftGUI a = GUI (\_ -> a)

instance EventSystem GUI where
   eventLoop handler = liftGUI (eventLoop handler)
   ...

instance WindowSystem GUI where
   createShellWindow title = liftGUI (createShellWindow title)
   ...

instance MutVars GUI where
   newRef init = liftGUI (newRef init)
   ...
```

The entire instance declarations for the `EventSystem` and `WindowSystem` type classes can be seen in Appendix B.2.4, and for the `MutVars` type class in Appendix B.2.1.

An application is composed of three main steps, firstly the state for the `GUI` monad must be initialised to specify default behaviour that will be provided by the windowing system. Secondly, the interface for the application must be created along with the specific behaviour

of the interface components. Finally, the event loop is started allowing events to be received and processed according to the behaviour specified by the previous steps. These steps are modelled by the startProg function:

```
startProg  :: GUI a -> IO ()
startProg w = do st <- newVar initGUIState
                 startingWithGUI st w
                 eventLoop (mainHandler st)
```

The interface is specified by the argument to the startProg function, and is used to create graphical components and add entries specifying their behaviour to the state encapsulated by the GUI monad. The initGUIState function returns an initial value for this state which defers all behaviour to the windowing system. The mainHandler function takes a reference to the state encapsulated in the GUI monad so that the response to an event for a particular window can be determined. The event handler determining the response is retrieved from the tables which form this state, by looking up the appropriate window and event type. This is performed by the processEvent function (see Section 2.5.2 for details of the lookupHandler function), which also passes the event to the retrieved event handler for processing:

```
mainHandler          :: GUIStateVar -> Event -> IO ()
mainHandler st event = startingWithGUI st (processEvent event)

processEvent       :: Event -> GUI ()
processEvent event = do handler <- lookupHandler event
                        handler event
```

Using the GUI monad, event handlers can be built as functions returning values of type GUI () rather than type IO (), thus allowing the event handlers to modify the tables describing the behaviour of the application. The family of coercion functions, such as handleLButtonDown need to be changed to work in the GUI monad:

```
type EventHandler = Event -> GUI ()

handleLButtonDown :: (Vector -> GUI ()) -> EventHandler
```

2.5.2 The GUI State

The behaviour of an application is stored in a table, indexed by values of type Window. This allows differing behaviours to be specified for each window. The entries in the table are tables themselves, indexed by the type EventType, with entries that are event handlers describing the actual responses:

```
type WindowTable   = Table Window EventHandlers
type EventHandlers = Table EventType EventHandler
```

We would like to use event processing functions, such as processLButtonDown as the entries in tables of type EventHandlers. However, different event processing functions have different types because the information they unpack from the event depends entirely upon the type of the event. By using functions like handleLButtonDown we can turn the event processing functions into event handlers which have one type and so can be used as entries in tables of type EventHandlers. We will, of course, need mechanisms to lookup and remove entries from such a table, for example:

```
main :: IO ()
main  = startProg simple

simple :: GUI ()
simple  = do window <- createShellWindow "Simple"
             showWindow True window
             addHandler window LButtonDown (ldown window)
             addHandler window Key (keydown window)

ldown         :: Window -> Event -> GUI ()
ldown window _ = setWindowCaption window "Left mouse button pressed"

keydown         :: Window -> Event -> GUI ()
keydown window _ = do removeHandler window LButtonDown
                      setWindowCaption window "Key pressed"
```

Figure 2.5: A simple application illustrating event handling

```
addHandler    :: Window -> EventType -> EventHandler -> GUI ()
removeHandler :: Window -> EventType -> GUI ()
lookupHandler :: Event -> GUI EventHandler
```

The `addHandler` function takes a window, an event type and a handler for these types of events, and modifies the state encapsulated by the `GUI` monad to include the new event handler. The `removeHandler` function takes a window and an event type, and modifies the state encapsulated by the `GUI` monad to remove the event handler for the specified event type. The `lookupHandler` function can be used to obtain the appropriate event handler for a given event passed to it as its argument. If no event handler exists for the event, then a reasonable course of action is to return the default event handler.

2.5.3 A Simple Event Handler

Figure 2.5 shows an example of using these functions to set up responses to particular events that alter the title text of a window when either the mouse is pressed or a key is pressed. When the mouse is pressed, this application changes the title text of the main window, however if a key is pressed then the title text is changed to reflect this and also the response to mouse clicks is changed so that nothing happens. This example illustrates how applications can handle multiple types of events, and also how the responses to these events can be changed as a program executes.

2.5.4 A Better Interface

Event handlers may modify the behaviour of the application through the use of the `addHandler`, and `removeHandler` functions. A slightly easier interface for the programmer can be constructed by defining specialised functions for defining the behaviour of an application in response to particular events. For example, to specify the behaviour required when the left mouse button is pressed down we can use the following function:

```
class WindowSystem m => DrawingSystem m where
   lineTo       :: Point -> m ()
   moveTo       :: Point -> m ()
   selectObject :: Object -> m Object
   deleteObject :: Object -> m Bool
   drawText     :: Point -> String -> m ()
   createPen    :: Int -> Colour -> m Object
```

Figure 2.6: Drawing primitives

```
onLButtonDown :: Window -> (Vector -> GUI ()) -> GUI ()
onLButtonDown window handler
   = addHandler window LButtonDown (handleLButtonDown handler)
```

Functions can be defined for other types of events in a similar manner. The other functions are shown in Appendix B.3.2, which also illustrates that in practice the argument of the `handler` function will expect more arguments than just a `Vector`. The arguments to the `handler` function depend upon the particular event.

2.6 Drawing Graphics

2.6.1 Graphic Primitives

Just as we introduced primitives for supporting event loops and basic window operations, we introduce appropriate primitives for drawing in windows. The primitives can be characterised by a type class that abstracts away from the monad the primitives are originally implemented in, similarly to the `EventSystem` and `WindowSystem` type classes. In the Embracing Windows framework, we opt to support only a handful of the possible operations that a windowing system supports for drawing graphics, and define the `DrawingSystem` type class to encapsulate them (Figure 2.6).

Instead of defining an instance of this type class for the `IO` monad, as we did for the `EventSystem` and `WindowSystem` type classes, we need to define a new monad, the `Draw` monad. The primitive drawing operations require a device context parameter not evident in the methods of the `DrawingSystem` type class, and this is exactly what will be supplied by the `Draw` monad. This method has been used in a declarative toolkit for programming GUIs using the ML language [19].

2.6.2 The Draw Monad

The process of drawing graphics in a window involves obtaining a device context, which encapsulates the environment for the drawing, such as the current colour, pen size and pen shape. It is cumbersome to have to obtain a device context before drawing, and furthermore the device context has to be passed to all of the primitive drawing operations. This can be avoided by using the same technique used to hide the details of the state storing the behaviour for events in the `GUI` monad. We can define a drawing monad, `Draw`, that hides the device

context being used. Since the drawing operations never directly modify the device context the type of monad we require is a state reader monad:

```
newtype Draw a = Draw (DC -> GUI a)

instance Monad Draw where
   return x     = Draw (\dc -> return x)
   Draw g >>= f = Draw (\dc -> do a <- g dc
                                  let Draw h = f a
                                  h dc)
```

The `Draw` monad supports operations that require a device context and that operate in the `GUI` monad. An operation in the `IO` monad can easily be handled by using the lifting function for the `GUI` monad, `liftGUI`. We are now in a position to declare an instance of the `DrawingSystem` type class for the `Draw` monad which will hide all of the details of device contexts. For each of the methods in the `DrawingSystem` type class we provide an implementation in terms of the appropriate primitive functions lifted from the `IO` monad to the `GUI` monad:

```
instance DrawingSystem Draw where
   lineTo (x, y)       = Draw (\dc -> lift (primLineTo dc x y))
   ...
```

The entire instance declaration can be seen in Appendix B.3.4. The `Draw` monad hides the details of the device context, but does not yet allow us to make use of this. We need a function that supplies the initial device context, equivalent to the value of the world supplied to a program written using the `IO` monad by executing the program:

```
startingWithDC             :: DC -> Draw a -> GUI a
startingWithDC dc (Draw d) = d dc
```

An application can draw in a window by simply obtaining an appropriate device context and then supplying this to the `startingWithDC` function along with a value of type `Draw a` describing the drawing to take place. For arbitrary drawing, we define the `drawInWindow` function which takes a reference to the window to draw inside of as its first argument. Its second argument defines the drawing to be undertaken, while its result is the corresponding value of type `GUI a` that will perform the drawing:

```
drawInWindow :: Window -> Draw a -> GUI a
```

The majority of windowing systems support the notion of a *paint* event, this is a special event issued by the windowing system when the contents of a window needs to be redrawn to keep the screen up to date. In the Embracing Windows system, we supply the following function that eases the task of writing code to respond to paint events:

```
onPaint :: Window -> (Window -> Draw ()) -> GUI ()
```

In response to a paint event, the `onPaint` function invokes its second argument to repaint the necessary window. The second argument defines the contents of a window as a function taking a reference to the window itself to a value of type `Draw ()`. This latter value defines the graphical content of the window.

A simple example of an application that draws a square with some text in the middle of it is listed in Figure 2.7. The `mkWindow` function creates a window that when closed will shut down the application by ending the event loop, its implementation uses the `createShellWindow` function and can be seen in Appendix B.3.2.

```
main :: IO ()
main  = startProg app

app :: GUI ()
app  = do window <- mkWindow "Simple GUI"
          showWindow True window
          onPaint window paintWin

paintWin       :: Window -> Draw ()
paintWin window = do drawsquare
                     drawText (20, 40) "Hello!"

drawsquare :: Draw ()
drawsquare  = do moveTo (10,  10)
                 lineTo (100, 10)
                 lineTo (100, 100)
                 lineTo (10,  100)
                 lineTo (10,  10)
```

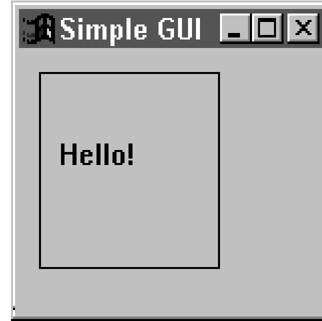


Figure 2.7: A simple drawing application

2.7 Controls

Common user interface components such as text labels, editable text fields, and push buttons are built-in to most windowing systems. In the majority of the existing windowing system interfaces for functional languages, these built-in components are eschewed in favour of building custom alternatives from scratch, instead the Embracing Windows framework provides an interface to such components. The main reason for ignoring built-in components is that the interface they provide is not usually orthogonal enough to integrate into a functional GUI development system. The building of components from scratch is also seen as a test of the expressiveness of a GUI development system. From a pragmatic viewpoint this has two disadvantages, firstly the components built from scratch are likely to differ slightly from their built-in counterparts and result in a non-standard look-and-feel. Secondly, the components built from scratch are likely to be less efficient than built-in components. Built-in components are often referred to as controls, and are windows with particular, predefined behaviour. In this report we only consider three kinds of controls, text labels, editable text fields, and push buttons, as the main principles can be illustrated with just these controls, although Windows 95 has many others.

Controls often have certain properties and behaviours in common. For example, all controls have some text associated with them such as the text of a button, label or edit field. Such commonalities between controls can be captured by using type classes. A general type class supporting operations for altering and retrieving the text of a control, and also for altering and retrieving information regarding the size of a control is defined as:

```
class Control a where
   setText :: a -> String -> GUI ()
   getText :: a -> GUI String
```

```
   setSize :: a -> Rect -> GUI ()
   getSize :: a -> GUI Rect
```

We provide an instance of this class for each kind of control, specifying the behaviour for the four operations as appropriate in each case. The instance declarations are shown in Appendix B.3.3.

2.7.1 Text Labels

A simple example of a control is a text label. The operations defined in the `Control` class are exactly the operations we require for a text label. By defining a data type and appropriate instance of the `Control` class, we can characterise text labels as controls:

```
data TextLabel = ...

instance Control TextLabel where
    ...
```

This allows us to manipulate text labels, but not to construct them. An appropriate construction function is required that will create an instance of the control built-in to the window system and return a handle that can be used to manipulate the control. The handle can be hidden from the programmer by encapsulating it inside of the `TextLabel` data type:

```
mkTextLabel :: String -> Window -> Rect -> GUI TextLabel
```

The first argument specifies the initial text of the text label, while the second and third arguments specify the window the text label is to be displayed in, and its position in this window. By using the `setText` and `getText` methods of the `Control` type class, we can now alter and retrieve the text of a text label control.

The internal operation of the control construction function, and the type class operations is specific to the windowing system being used. Section 2.9.2 gives details on an implementation for the Windows 95 system.

2.7.2 Edit Fields

Edit fields can be characterised as controls in the same way as text labels. There are some extra operations that edit fields support as well as the ones from the `Control` type class. Any editable control supports two operations, the first specifying the behaviour of the control when its content has been changed, the second specifying the behaviour when the content of the control is committed. For an edit field, the content changes whenever the user edits the text in the edit field. The content is considered to be committed when the user presses the return key. The semantics for committing the content of a control depends upon the type of the control. For example, a listbox may commit its current selection when the input focus is switched away from the control, but a group of radio buttons may commit whenever the selected button is changed. In a similar method to capturing the basic operations for all controls in the `Control` type class, we can capture the notion of change and commit in a type class:

```
class Control a => Editable a where
   onChange :: a -> GUI () -> GUI ()
   onCommit :: a -> GUI () -> GUI ()
```

The first parameter to the onChange and onCommit functions determines the control to specify behaviour for, while the second parameter specifies the behaviour.

We can now define a data type for edit fields, with suitable instances of the Control and Editable type classes, and also a construction function:

```
data EditField = ...

instance Control EditField where
    ...

instance Editable EditField where
    ...

mkEditField :: String -> Window -> Rect -> GUI EditField
```

Editable controls maintain state, and common operations on this state will be to retrieve it, and to modify it. We can define functions for these operations, however their types depend not only upon the type of the control being manipulated but also upon the type of the state. To model this we would need to use a multiple parameter type class. The extended definition of the Editable type class would be:

```
class Control a => Editable a s where
   onChange :: a -> GUI () -> GUI ()
   onCommit :: a -> GUI () -> GUI ()
   setState :: a -> s -> GUI ()
   getState :: a -> GUI s
```

Haskell 1.3 does not support multiple parameter type classes so we must be content to implement the setState and getState operations on a case by case basis for each type of control.

2.7.3 Buttons

Most windowing systems support a variety of buttons, such as push buttons, radio buttons, and check boxes. All of these types of button maintain a current state, which is editable by the user interacting with the button. Buttons are editable controls and, as such, we need only define an appropriate data type for each type of button and supply instances of the Control and Editable type classes. For a simple pushbutton that is used to acknowledge an action, a single state button, then the onChange function can be used to specify the behaviour resulting from the single action of pushing the button:

```
data PushButton = ...
instance Control PushButton where
   ...

instance Editable PushButton where
   ...

mkPushButton :: Window -> String -> Rect -> GUI PushButton
```

```
main :: IO ()
main  = startProg counter

counter :: GUI ()
counter  = do window <- mkWindow "Counter"
              display <- mkTextLabel window "0" display_rect
              button <- mkPushButton window "Increment" button_rect
              onChange button (increment display button 0)
              showWindow True window

increment :: TextLabel -> PushButton -> Int -> GUI ()
increment display button count
   = let count' = count + 1
     in do setText display (show count')
           onChange button (increment display button count')

display_rect = ( (0, 0),  (100, 30) ) :: Rect
button_rect  = ( (0, 35), (100, 30) ) :: Rect
```

Figure 2.8: The counter application

2.8 The Counter Program

A common example used to illustrate how to build a simple GUI is the counter program. This program has a button labelled "Increment" and a text field that displays a number. When the user presses the button, the value displayed in the text field is incremented by one. Using the mechanisms described in the previous sections, we can implement a counter application, using the definitions in Figure 2.8.

The application consists of three functions:

- `main` makes use of the `startProg` function to create the application's main window, graphical components and start the event loop.
- `counter` creates the graphical components of the application: the display and the push button. It also sets the behaviour of the push button when pressed to be specified by the `increment` function.
- `increment` handles the event of pushing the button, it takes the current counter value as its third argument and increments it to get the new counter value. The text of the display is updated with the new value, and finally the behaviour of the button is altered to reflect the new counter value.

The two auxiliary functions, `display_rect` and `button_rect` specify the position and size of the display and push button with respect to their containing window. The state required by this application, that is the value of the counter, is stored in the actual event handler itself. The ability to dynamically manipulate the event handlers specifying the application's behaviour is illustrated here. If we could not update the event handler determining the response to button presses, then we would have to store the state using a different mechanism.

2.9 Implementation Details

2.9.1 Events

The Embracing Windows framework interfaces to Windows 95, in which an event is defined as a 4-tuple, with access functions to the components of the tuple defined in the obvious way:

```
type Message = Int
type Event   = (Window, Message, Int, Int)

getWindow  :: Event -> Window
getMessage :: Event -> Message
getWParam  :: Event -> Int          -- 3rd component of tuple
getLParam  :: Event -> Int          -- 4th component of tuple
```

The first component of this 4-tuple is a reference to the window that the event is associated with, the second component specifies the type of the event, such as a mouse click. The last two components contain extra information specific to the type of the event.

The types of events are defined by the `EventType` datatype, which has a number of data constructors, one for each different type of event. The `getEventType` function converts a value of type `Event` into one of type `EventType` by examining the second component of the 4-tuple comprising the event. The details of the `EventType` datatype and the `getEventType` function can be seen in Appendix B.2.3.

2.9.2 Controls

A first attempt to provide an interface to controls might be to add a more general window creation primitive that takes an extra argument determining the type of the window to be created. This could then be used as the basis of a number of different functions that create specific types of controls, such as a button:

```
createButton                   :: Window -> String -> Rect -> IO Window
createButton parent title rect = mkControl "button" parent title rect
```

The `mkControl` primitive creates a new control of a specific type. The first argument determines the type of the control, e.g. `"button"`, `"edit"`, `"static"`, whilst the second argument specifies the parent window for the control. The remaining arguments detail the text associated with the control, and its size and position. However, controls also need to be able to communicate events to the application. For example, when a button is pressed, the application might want to respond in a particular way. In Windows 95, a control belongs to the window it is displayed inside of, the parent window. Any events that the control wishes to communicate to the application are sent as *notification* events to the parent of the control.

Unfortunately, this means that a parent window must know about all of its contained controls so that it can respond appropriately when it receives notification of an event from one of them. It is quite common for the control itself to be able to handle the response, and this can be achieved by wrapping a control inside a transparent parent window that is exactly the same size as the control. This wrapped control is the component that is used in a GUI application. When a control receives an event, this will be communicated to the transparent parent window which processes the event easily as it knows that it can only have come from one control. A generic primitive for creating transparent parent windows, `mkChildWindow`, can be used to encode these wrapped controls within the functional language. The previous example for creating a button control now becomes:

```
mkPushButton                  :: Window -> String -> Rect -> GUI Window
mkPushButton parent text rect = do window <- mkChildWindow parent rect
                                   button <- createButton window text rect
                                   return (PushButton window button)
```

The `mkChildWindow` function creates a new window that is a child of the window specified by its first argument. The second argument determines the size of the child window. The other control creation functions, `mkEdit` and `mkLabel` can be defined in a similar fashion, and are shown in Appendix B.3.3.

Chapter 3

Widgets

Using the framework described in Chapter 2, we can implement an abstract interface for building GUI's. This interface has similarities to both the Haggis System [6] and the Tk-Gofer System [25]. The main concept in this interface is the widget, which is a description of a graphical object with a specific behaviour and appearance. Widgets can be composed to create other widgets using combinators. A library of predefined widgets provides support for text labels, edit fields and buttons.

3.1 The Representation of Widgets

A widget is represented as a monadic value, thus allowing the `return` and `>>=` operations of the monad to be used to combine widgets. A widget is just a description of how to create a graphical component with specific behaviour and layout. The monad used to represent widgets is defined as[1]:

```
newtype Widget a = Widget (Window -> GUI a)

thenW       :: Widget a -> (a -> Widget b) -> Widget b
m `thenW` f = Widget (\window -> do let Widget m' = m
                                    r <- m' window
                                    let Widget n' = f r
                                    r' <- n' window
                                    return r')


returnW  :: a -> Widget a
returnW x = Widget (\window -> return x)
```

The data constructor `Widget` takes a value of type `Window -> GUI a` as its argument, and this value can be thought of as the widget's realization function. It takes an argument identifying the window in which the widget is to be realized, and creates the widget, returning an appropriate value such as a handle that can be used to further manipulate the widget. A program written using widgets makes use of the function `wopen` which realizes a widget in a top level window. The implementation of `wopen` can be seen in Appendix B.4.1, and its type is:

[1] The definition in Appendix B.4.1 also takes layout into account

```
wopen :: String -> Widget a -> GUI Window
```

`wopen` creates a new top level window with a title specified by its first parameter, and the widget described by the second parameter is realized inside this window. A reference to the new top level window is returned as the result of the whole action.

3.2 A Library of Widgets

In the current implementation of the Embracing Windows system, there are three predefined widgets provided. Button widgets are created using `buttonW`, with arguments that specify the text of the button, size of the button, and a value of type `GUI ()` specifying the action to be performed when the button is pressed. Similarly, edit fields can be created using the `editW` function with arguments specifying the initial text and size of the edit field. The value returned by an edit field can be used to retrieve the text of the edit field using the `getText` function of the `Control` type class. Text labels can be created using the `textW` function with arguments specifying the text and size of the label:

```
buttonW :: String -> Vector -> GUI () -> Widget PushButton
editW   :: String -> Vector -> Widget EditField
textW   :: String -> Vector -> Widget TextLabel
```

3.3 Composite Widgets

The monadic operations `return` and `>>=` form a basis for building composite widgets. The *do notation* described in Section 1.2 can be used to make the syntax used in defining composite widgets clearer and more concise. A composite widget that combines a label with an editable text field can be defined as:

```
labelledEditW       :: String -> Widget EditField
labelledEditW label = do textW label (100, 30)
                         editW "" (100, 30)
```

This widget illustrates a useful abstraction, gluing a text label to an arbitrary widget. Because we are using a higher order functional language, it is easy to define a combinator capturing this abstraction:

```
labelW              :: String -> Widget a -> Widget a
labelW label widget = do textW label (100, 30)
                         widget
```

Unfortunately the size of the text label widget must be explicitly stated, and this makes the abstraction less useful. However, it is an easy extension of the layout system to incorporate automatic sizing of library widgets, such as text labels, based on the content of the widget. If this extension is made then the `labelW` abstraction becomes more useful.

3.4 Layout of Widgets

The monadic widget combinators only specify the behavioural composition of widgets. The relative layout of widgets is specified separately from the behavioural composition by using

primitives based on TEX's [12] layout mechanism. Two layout primitives are provided, `hbox` and `vbox`, with which the layout of widgets can be specified (or with which other layout combinators can be built). When widgets are combined using do notation, one can think of the widgets as being piled on top of each other. The `hbox` layout combinator pulls these piled widgets out to form a horizontal row of widgets, while `vbox` pulls the piled widgets out to form a vertical column of widgets. If no layout combinator is used then the z-order of widgets piled on top of each other is determined by the order in which the widgets were combined by the use of the do notation. The basic primitives for specifying layout are essentially the same as those provided in the Haggis system. The Embracing Windows framework supports a simple form of TEX's notion of glue, called space widgets. A space widget occupies screen space but has no behaviour:

```
hspace :: Int -> Widget ()
vspace :: Int -> Widget ()
```

The argument to these functions determines the amount of space that the widgets will take up, and is specified in pixels. A possible improvement would be to use a form of device independent units.

Using these functions, we can define new layout combinators for putting margins around widgets. Placing a margin to the left and right of a widget leads to an `hmargin` layout combinator:

```
hmargin                    :: Int -> Widget a -> Widget a
hmargin margin_width widget = hbox (do hspace margin_width
                                       result <- widget
                                       hspace margin_width
                                       return result)
```

Here we can see how a composite widget is built, by combining a space widget, the real widget, and another space widget. These three widgets are layed out horizontally using the `hbox` layout combinator. A similar function, `vmargin :: Int -> Widget a -> Widget a`, lays out a widget with a margin above and below it. By combining these two layout functions, a third layout combinator, `margin` can be created that puts a margin all the way around a widget:

```
margin      :: Int -> Widget a -> Widget a
margin width = hmargin width . vmargin width
```

Revisiting the labelled text widget example described in Section 3.3, we can include, in the combinator definition, the use of a layout combinator to constrain the label to be placed at the left of the widget, with a space between the label and the widget:

```
improvedLabelW             :: String -> Widget a -> Widget a
improvedLabelW label widget = hbox (do textW label (100, 30)
                                       hspace 30
                                       widget)
```

Currently the layout system requires that the size of library widgets is specified explicitly rather than inferred in some way. For example, a text label's initial size can be determined from the length of the text initially displayed. However, this was not implemented in the Embracing Windows framework so as not to complicate the system. A disadvantage of this is that it makes widgets less reusable, as explicit sizes have to be encoded in the definitions.

3.4.1 Implementation of Layout

The implementation of the layout of widgets involves two main steps:

- Each widget has associated with it a preferred size and this information is passed to the containing widget, which may be specified by one of the layout primitives. The containing widget uses this information to determine the size it would like, and this is again passed up the hierarchy of widgets. Eventually the root widget is reached at which point the requested size of all widgets has been collated to form an overall request for the application.

- The application may well have been allocated a set amount of screen space, and this may not match the requested size of the root widget. The actual space available is passed to the root widget, which partitions it out between its child widgets according to the space they have each requested. This process continues, with the available size for widgets propagating back down the widget hierarchy. Eventually, a widget that has no children will be reached, and this is the point at which any change in size of the widget will occur.

This process is implemented by modifying the representation of widgets to include screen space requests, and also functions to realize a change in size of a widget. The widget composition combinators pass screen space requests up the hierarchy by combining the many requests of their children to form their own request. When a widget is realized in a window, then the requested size is used to set the initial size of the new window. The appropriate sizing of widgets is accomplished by using sizing functions belonging to each widget in the hierarchy of widgets. The widget composition combinators form their own sizing functions from logic determining how to split their allocated space amongst their children and the sizing functions of the children themselves. When a widget is resized, the actual size allocated to the widget is propagated down the widget hierarchy through the sizing functions.

This mechanism is further complicated by widgets that are willing to accept changes in their size, thus making the logic that splits up allocated space between widgets more involved. Basically, if a widget can change in size, then its size will be changed in preference to a widget that has asked for a fixed amount of screen space. The details of the algorithms used for layout can be seen in Appendix B.4.2.

3.5 Stateful Widgets

We are using widgets to model objects in a graphical user interface, and quite often such objects will require state. Mutable variables can be used to give widgets access to state, but, if not used carefully, widgets can become hard to reuse. A widget version of the counter application can be built using a mutable variable to store the current value of the counter, as shown in Figure 3.1. The mutable variable is created by the `stateW` function, which is a lifting of the standard `newRef` function into the `Widget` monad. `stateW` creates a stateful widget that has no on screen representation or behaviour, but when created returns a reference to a piece of mutable state. Notice that the state required by the counter is entirely encapsulated inside of the counter widget by the use of the `stateW` function.

A common use of the `stateW` function is to create a piece of state to store the contents of a text field, and this can be abstracted out into a stateful text field:

```
main :: IO ()
main  = startProg w
        where w = do wopen "Counter" (counter 0)
                     return ()

counter :: Int -> Widget PushButton
counter init
   = vbox (do st <- stateW init
              display <- textW (show init) display_size
              buttonW "Increment" button_size (handler display st))

handler           :: TextLabel -> Ref Int -> GUI ()
handler display st = do count <- getRef st
                        let count' = count + 1
                        setRef st count'
                        setText display (show count')

display_size = (100, 30) :: Size
button_size  = (100, 30) :: Size
```

Figure 3.1: A Widgets counter

```
stateTextW :: Show a => Size -> a ->
                        Widget (TextLabel, ((a -> a) -> GUI ()))
stateTextW size init
   = do st <- stateW init
        label <- textW (show init) size
        let update f = do v <- getRef st
                          let v' = f v
                          setRef st v'
                          setText label (show v')
        return (label, update)
```

The `stateTextW` function returns an action whose result is a pair of values, the first of which can be used in operations on the label, while the second is a function that can be used to alter the state of the label. The stateful text field can be reused easily, and the improved counter program is listed in Figure 3.2.

The main advantage of this program over the one in Section 2.8 is compositionality. It allows the counter application to be constructed as a single widget that could easily be used in other applications by using the widget combinators `>>=` and `return`. Also the automatic layout support simplifies the physical description of the GUI, with only the width and height of widgets needing to be specified.

3.6 Comparison to Haggis

The syntax for writing Widget programs is very similar to that used by Haggis but, unlike Haggis, the Widget system does not make any use of concurrency. To illustrate the difference that concurrency makes to specifying GUIs, we present a Haggis counter application

```
main :: IO ()
main  = startProg w
        where w = do wopen "Counter" (counter 0)
                     return ()

counter :: Int -> Widget PushButton
counter init
   = vbox (do (display, update) <- stateTextW display_size 0
              buttonW "Increment" button_size (update (+1)))

display_size = (100, 30) :: Size
button_size  = (100, 30) :: Size
```

Figure 3.2: An Improved Widgets counter

```
main :: IO ()
main  = do env <- mkDC ["*title:Counter"]
           (lbl, lbl_dh) <- label "0" env
           (btn, button_dh) <- button (text "Increment") (+1) env
           forkIO (handler 0 btn lbl)
           realiseDH env (vbox [lbl_dh, button_dh])

handler          :: Int -> Button (Int -> Int) -> Label -> IO ()
handler n btn lbl = do f <- getButtonClick btn
                       let n' = f n
                       in setLabel lbl (show n')
                       handler n' btn lbl
```

Figure 3.3: A Haggis counter

(Figure 3.3) and contrast it with the Widget version.

We will not explain all of the functions used in the Haggis counter application, but refer the interested reader to the documentation included in the Haggis system [5]. The Widget and Haggis counters both consist of a combination of a text field and a button. The Haggis counter does not specify the behaviour of the button using a function that is called in response to user input. Instead, a separate thread of control is started using the `forkIO` function. This separate thread waits for the button to be pressed and then applies the integer modifying function emitted by the button to the current value of the counter. The text label is altered to show this new value, and finally we return to the beginning of the `handler` function to wait again for another button press. The `handler` function for Haggis is similar to the `handler` function for Widgets, except that, in the widget program, we don't need to explicitly wait for a button press. Instead, in the widget program we specify the behaviour required when a button press event occurs by the use of a function from the type of an event to the type of an I/O action (in this case the type is `GUI ()`). Similarly, when we have processed a button press, instead of direct recursion we return to the event loop to await the next event.

The addition of concurrency in the Haggis program frees the programmer from having to deal with the event loop. However, in our example, we have written a mini event loop that polls for button presses. In general, the Haggis programmer may end up writing a number of smaller more specific event loops. An application whose responses to events changes as the program executes will be better suited to the Haggis system. For example, an application that has two buttons which, when pressed in a particular sequence, quit the program, requires the code describing the behaviour of the application to be split up into two handlers:

```
main :: IO ()
main  = startProg w
        where w = do wopen "Test" testW
                     return ()

testW :: Widget PushButton
testW  = vbox (do one <- buttonW "One" button_size (return ())
                  buttonW "Two" button_size (onChange one quitapp)

button_size = (100, 30) :: Size
```

Initially, pressing the first button has no effect, however, pressing the second button changes the behaviour of the first button such that when it is pressed the application will be shut down.

In the Haggis system, we do not need to split up the description of the application's behaviour, instead it can be described as one piece of code:

```
main :: IO ()
main  = do env <- mkDC ["*title:Test"]
           (one, one_dh) <- button (text "One") () env
           (two, two_dh) <- button (text "Two") () env
           forkIO (handler one two lbl)
           realiseDH env (vbox [one_dh, two_dh])

handler :: Button () -> Button () -> IO ()
handler one two = do getButtonClick two
                     getButtonClick one
                     shutdownShop
```

The `handler` function waits until the button labelled "Two" is pressed, and then continues by waiting until the button labelled "One" is pressed. Once this has happened, the application is shut down. If we had a large number of buttons that had to be pressed in a particular sequence then it would be cumbersome to express this in the Widget system, however it would be relatively easy to express in the Haggis system.

If the behaviour for each of the buttons is independent then the program written in the Widget system would be very similar to the program written in the Haggis system. Both programs would create two buttons, and have two separate functions specifying the behaviour of the buttons independently. In the Haggis version, two threads of control would be concurrently forked that wait for button presses and perform the appropriate behaviour, whilst in the Widget version, two handler functions specify the behaviour of the buttons and are called in response to user input.

Chapter 4

Fudgets

4.1 Overview of Fudgets

The Fudgets system [9] is a well established toolkit for building GUI applications in the functional language Haskell. It uses an abstraction, the fudget, to describe a self contained GUI component.

A fudget is built on top of the concept of a stream processor. A stream processor is a process with one input stream and one output stream. The type of a stream processor that inputs values of type `a` and outputs values of type `b` is written as `SP a b`. Stream processors are built from three basic stream processors using a continuation passing style. The basic stream processors are:

```
getSP  :: (a -> SP a b) -> SP a b
putSP  :: a -> SP b a -> SP b a
nullSP :: SP a b
```

The `getSP` stream processor retrieves a value from the input stream and processes it using the first argument to `getSP`, transforming itself into a new stream processor. Similarly, `putSP` outputs the value indicated by its first argument on the output stream and turns into the stream processor specified by its second argument. The `nullSP` stream processor terminates immediately ignoring any values on its input stream and producing no values on its output stream.

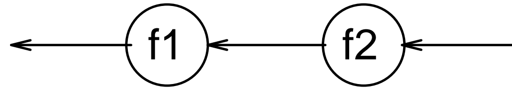
A simple example of a stream processor is the `mapAccumlSP` function, which creates a stream processor with an internal state. The first parameter to this function specifies a state modifying function, that changes the state given the value on the stream processor's input stream. This state modifying function also generates the values emitted on the output stream. This stream processor can be written in terms of the basic stream processors as:

```
mapAccumlSP         :: (s -> b -> (s, c)) -> s -> SP b c
mapAccumlSP f state = getSP $ \input ->
                      let (state', output) = f state input
                      in putSP output (mapAccumlSP f state')
```

A fudget is just a stream processor that can communicate with the windowing system. Fudgets can be combined using combinators that specify how the single input and output streams of fudgets are connected together. These combinators are, `>==<`, `>*<` and `>+<` which compose fudgets in series, and in parallel (either untagged or tagged), respectively. The behaviour of these combinators is defined as follows:

- `>==<`, composes two fudgets serially with the output from the first fudget sent to the input of the second:

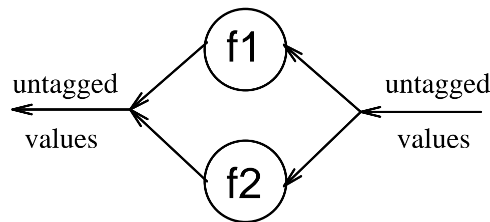
```
>==< :: F a b -> F c a -> F c b
```



f1 >==< f2

- `>*<`, composes two fudgets in parallel. The input values are routed to both fudgets, and the output values are merged to form the output stream:

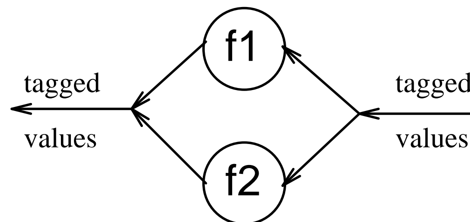
```
>*< :: F a b -> F a b -> F a b
```



f1 >*< f2

- `>+<`, composes two fudgets in parallel. The input values are expected to be tagged indicating to which fudget the value is to be routed. Similarly, the output values are tagged indicating which fudget they came from:

```
>+< :: F a b -> F c d -> F (Either a c) (Either b d)
```



f1 >+< f2

The counter example can be expressed with fudgets using the code in Figure 4.1, which is a simpler version of the *SmallCounter.hs* example from the original fudgets distribution [3]. The `fudlogue` function takes the main application fudget as its argument and turns it into the appropriate type for the Haskell I/O system. A fudget is realized in a top level or shell window using the `shellF` function. This function takes two arguments, the first is a string specifying the title for the shell window, and the second is the fudget to be realized inside the shell window. In this example, the `counterF` fudget combines three separate fudgets using the series combinator, `m >==< n`. The `intDispF` fudget creates an integer display, while the

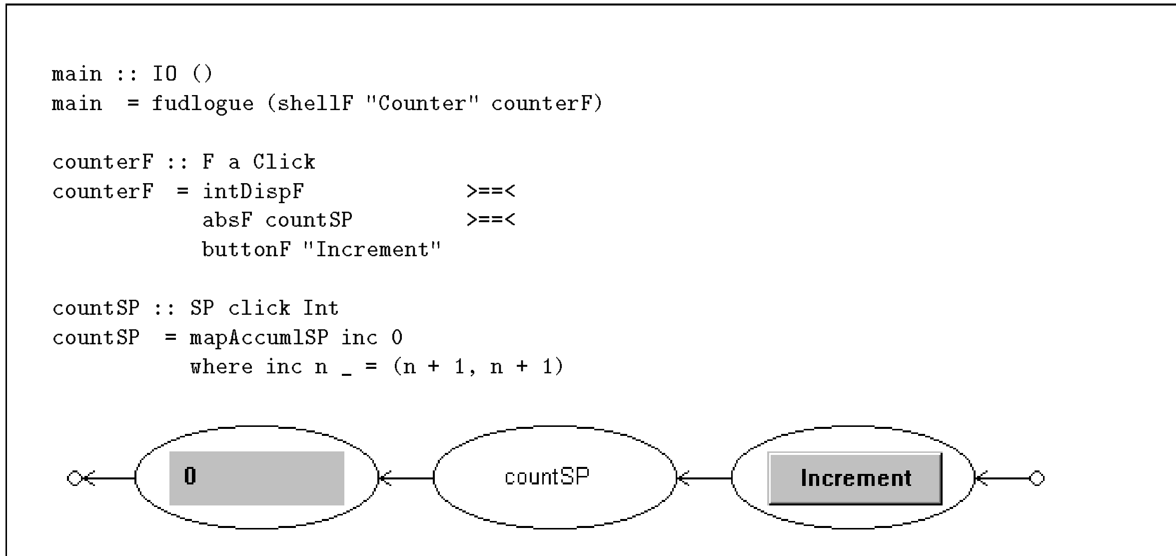


Figure 4.1: A Fudget counter

`buttonF` fudget creates a labelled button. The fudget connecting the button to the integer display is an abstract fudget; it has no input or output to the window system, and only communicates with the outside world through the fudgets it is linked to. Abstract fudgets are specified using stream processors, with the `absF` function converting a stream processor into a fudget:

```
absF :: SP a b -> F a b
```

A fudget is a stream processor that can also communicate with the windowing system, and so the `absF` function merely turns a stream processor into a fudget that acts like the stream processor but does not communicate with the windowing system.

4.2 Implementation of Fudgets

The original implementation of the Fudget system used stream processors as the basic building blocks for describing process networks, extending them with connections to the window system to create fudgets. This method could be adopted to encode the Fudget system in the Embracing Windows framework. However, it does not lend itself well to encapsulating existing GUI components such as the controls in Windows 95 as fudgets. An alternative approach based on work by Reid and Singh[21], uses a functional representation for fudgets:

```
type Handler a  = a -> GUI ()
type Fudget a b = Window -> Handler b -> GUI (Handler a)
```

Here, a fudget is modelled as a function returning a realization action that will create the appropriate GUI component. The function requires the window in which the fudget is to be realized as its first argument, whilst the second argument is an output handler, which will be used to send output to another fudget. The return value is of type `GUI (Handler a)` indicating that the realization action may perform I/O, and returns an input handler that can be used to send input to this fudget.

Using this encoding, it is possible to create atomic fudgets for buttons, edit fields, and text labels. For example, a button fudget is created by using the `buttonF` function,

```
buttonF :: String -> Size -> F Click Click
buttonF text (w, h) parent outputHandler
   = do button <- mkPushButton parent text ((0, 0), (w, h))
        onChange button (outputHandler Click)
        return inputHandler
         where inputHandler a = outputHandler Click
```

The `mkPushButton` control construction function is used to create the window's push button control. The `onChange` function is used to set the behaviour of the button when it is pressed. In this case when the button is pressed the value `Click` is emitted on the output stream of the fudget. Finally the input handler returned simulates the button being pressed by emitting the value `Click` on the output stream of the fudget.

The fudget combinators simply map onto functions that plumb together the input and output handler functions of fudgets. For example, the definition of the `>==<` combinator follows directly from the representation we are using for fudgets, connecting together the input handler of the second fudget to the output handler of the first fudget:

```
(f1 >==< f2) parent outputHandler
   = do handler      <- f1 parent outputHandler
        inputHandler <- f2 parent handler
        return inputHandler
```

The other combinators, `>*<` and `>+<` are expressed similarly and can be seen in detail in Appendix B.5.7.

Stream processors are simply fudgets that do not communicate with the windowing system. The representation used for fudgets can be reused for stream processors by using a type synonym:

```
type SP a b = F a b
```

The basic stream processors can easily be encoded using the functional representation used for fudgets, for example, we can code the `getSP` stream processor as:

```
getSP :: (a -> SP a b) -> SP a b
getSP f parent outputhandler
   = let inputhandler a = do f a parent outputhandler
                             return ()
     in return inputhandler
```

The `getSP` stream processor does not create any windows but simply returns an input handler. This handler invokes the continuation specifying the behaviour of the stream processor being created given the value read from the input stream.

4.3 Layout of Fudgets

The fudget combinators, `>==<`, `>*<`, and `>+<`, can be used to link together fudgets behaviourally. However, they do not specify any details about the layout of the fudgets being combined. The original implementation of the Fudgets system provided support for three ways to layout fudgets:

- **Placer Layout:** This method uses functions that modify the layout of a single fudget. Because fudgets can be combined using the fudget combinators, this may alter the layout of many fudgets.
- **Combinator Layout:** This method uses variants of the fudget combinators to specify the layout of a fudget program. The flexibility in the layouts possible is constrained by the flow of data in the fudget program because layout is based on fudget combinators that control the flow of data.
- **Name Layout:** This method specifies the layout of fudgets independently from the specification of the flow of data between fudgets. Fudgets are named, and the layout specified in terms of these names, resulting in a more flexible mechanism than combinator layout.

In the Embracing Windows framework, only the first and second of these methods for layout has been implemented. In the original Fudgets system, combinator layout is implemented in terms of placer layout, and this approach is taken in the Embracing Windows framework.

All of the above methods for specifying the layout of a fudget program do so in a hierarchical fashion, with each fudget residing in a box. These boxes can be placed together using placers. As in the Widgets system, placers are based upon the box mechanisms of TEX. Examples of some placers are:

```
horizontalP :: Placer
verticalP   :: Placer
```

The `horizontalP` placer lays out a group of fudgets next to each other horizontally, while the `verticalP` placer lays out fudgets vertically. Using a placer, the layout of certain fudgets can be specified by using the `placerF` function:

```
placerF :: Placer -> F a b -> F a b
```

This function applies the placer to all of the fudgets composing the single fudget specified by the second argument. Revisiting the counter example, we can layout the fudgets vertically by using `placerF`:

```
improvedcounterF :: F a Click
improvedcounterF  = placerF verticalP counterF
```

Some useful layout functions defined in terms of the `placerF` function are:

```
hBoxF :: F a b -> F a b
hBoxF  = placerF horizontalP

vBoxF :: F a b -> F a b
vBoxF  = placerF verticalP
```

The fudget combinators for layout take a tuple describing a fudget and a layout orientation as their first argument, followed by another fudget as their second argument. The two fudgets are combined using the normal fudget combinators, but are also placed relative to each other according to the orientation specified.

```
main :: IO ()
main  = fudlogue (shellF "Counter" counterF)

counterF :: F a Click
counterF  = (intDispF display_size, Above) >==#<
            (absF countSP >==< buttonF "Increment" button_size)

countSP :: SP click Int
countSP  = mapAccumlSP inc 0
           where inc n _ = (n + 1, n + 1)

display_size = (100, 30) :: Size
button_size  = (100, 30) :: Size
```

Figure 4.2: A Fudget counter with layout

```
data Orientation = Above | Below | RightOf | LeftOf

>#==< :: (F a b, Orientation) -> F b c -> F a c
>#*<  :: (F a b, Orientation) -> F c d -> F (Either a c) (Either b d)
>#+<  :: (F a b, Orientation) -> F a b -> F a b
```

The counter example can now be written using these combinators as can be seen in Figure 4.2. The implementation of layout for fudgets follows the same process as for Widgets, with requests for screen space propagating up through a hierarchy of fudgets, and the actual allocated space propagating down the hierarchy through functions that resize fudgets.

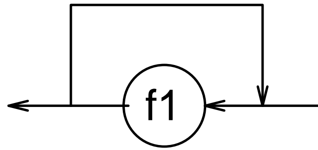
Similarly to the widget system, explicit sizes are required for the library fudgets such as push buttons, editable text fields, and text labels. In the original fudgets system library fudgets size themselves according to the content of the fudget. For the same reason as in the widget system we have not implemented this ability to automatically size library fudgets depending upon their content.

4.4 Looping Combinators

A number of combinators from the original implementation of the fudgets system have been left out from the implementation using the Embracing Windows framework. Some of these combinators cannot be encoded due to the choice of representation for fudgets. In particular the representation does not preserve the correct order of input values arriving at a fudget.

This implies that certain recursive combinators cannot be supported. For example, consider the `loopF` combinator, which connects the output of a fudget back to its own input, creating a feedback loop:

```
loopF :: Fudget a a -> Fudget a a
```



loopF f1

The output from the fudget provided as an argument to this combinator forms the output of the composed fudget, but is also sent back to the fudget's input. Using a fixpoint combinator, we can attempt to implement such a looping combinator as:

```
loopF :: F a a -> F a a
loopF f parent outputHandler
   = let inputhandler = fix (\handler ->
                               f parent (\x -> do outputHandler x
                                                  handler x))
     in return inputhandler
```

Here the `fix` function is a fixpoint combinator lifted into the `GUI` monad. The primitive fixpoint combinator is implemented in the `IO` monad, and the `WindowSystem` type class can be used to lift the primitive into different monads. The type of the fixpoint combinator for a monad m, is:

```
fix :: (a -> m a) -> m a
```

This primitive allows us to use the input handler returned by the fudget we are actually constructing, and hence we can route a value back to the input of the fudget that produced the value.

However, this does not work as we might hope; when a value is produced as output from a fudget composed using the `loopF` combinator it is *immediately* piped around to the fudget's input and processed, potentially before any values that have already been sent to the fudget on its input stream. Input values are thus processed out of order, unlike the original Fudgets system. This problem is important as most reasonable applications will require a use of such looping combinators at some point. Purpose built combinators could be used to ensure a particular order of processing of input values, or alternatively we could revert to using a representation of fudgets based on stream processors as in the original Fudget system implementation.

Because stream processors are implemented as fudgets that do not make use of the windowing system, they also suffer from these problems. In particular the `mapAccumlSP` function used in the counter example for maintaining state cannot be written using the basic stream processors, and is instead implemented using mutable state. The problem is that the `mapAccumlSP` function maintains state by using a feedback loop to feed the latest version of the state back into itself. Using mutable state, this function can be written as:

```
mapAccumlSP :: (a -> b -> (a, c)) -> a -> SP b c
mapAccumlSP f init parent outputhandler
   = do state <- newRef init
        return (inputhandler state)
      where inputhandler state a = do s <- getRef state
                                      let (s', b) = f s a
                                      setRef state s'
                                      outputhandler b
```

This stream processor creates a mutable variable to store the internal state in, and returns an input handler. This handler is invoked when a value is read from the input stream. The state is extracted from the mutable variable and modified according to the function specified as the first argument to the `mapAccumlSP` function. The new state is stored in the mutable variable, and a value emitted on the output stream.

Chapter 5

The Essence of Functional GUI's

In this Section, we present an essence of the two high-level systems developed in Chapters 3 and 4. A mapping between components of the two systems is described which can allow a mixture of the two systems within individual applications.

The essence of the Fudgets system can be seen in Figure 5.1, and the essence of the Widgets system in Figure 5.2. These two systems are similar, in that they both have:

- A datatype representing graphical components.
- A number of atomic graphical components.
- Combinators for building complex graphical components from the atomic ones.

The major difference between the two systems is in the combinators used to build composite graphical components. In the fudget system, the combinators specify the relationships between the input and ouput values produced by the components when the GUI is used. In the widget system, the combinators are the monadic operations `returnW` and `thenW`. The atomic components return values that can be used to manipulate the component, and are often referred to as handles. A dependency between components involves binding a name to the handle associated with one of the components and then making use of this in the specification of the related component. Since both systems use the combinators to express relationships between graphical components, it is not surprising that we can define a mapping between the two systems, albeit a restricted one. This mapping allows graphical components

```
data Fudget a b = ...

buttonF  :: String -> F Click Click
intDispF :: F Int a
labelF   :: String -> F String a

(>==<) :: F a b -> F b c -> F a c
(>*<)  :: F a b -> F a b -> F a b
(>+<)  :: F a b -> F c d -> F (Either a c) (Either b d)
```

Figure 5.1: The essence of Fudgets

```
data Widget a = ...

buttonW :: String -> Size -> GUI () -> Widget PushButton
editW   :: String -> Size -> Widget EditField
textW   :: String -> Size -> Widget TextLabel

thenW   :: Widget a -> (a -> Widget b) -> Widget b
returnW :: a -> Widget a
```

Figure 5.2: The essence of Widgets

from one system to be used in a limited way in the other system. The restricted fudget to widget mapping is accomplished by the `fudgetToWidget` function:

```
fudgetToWidget  :: F a b -> Widget ()
fudgetToWidget f = Widget (\window -> do f window nullHandler
                                         return ())

nullHandler  :: Handler ()
nullHandler _ = return ()
```

The `nullHandler` function is a simple output handler that is used to process any output produced by the fudget. The input handler that the fudget returns as a result of its realization is ignored. This limits the usefulness of the mapping because the wrapped fudget cannot interact with other widgets. The mapping could be used in an accounting application that provides an option to start a calculator in a separate window for independant calculuations. If a fudget version of a calculator is available then the rest of the accounting application could be written using the Widgets system with the option for displaying the calculator using the fudget version of the calculator under the above mapping. Since there is no interaction between the calculator and the accounting application the above mapping suffices. In general it would be useful to have a stronger mapping that allowed the fudget to interact with other widgets.

Using a similar method, we can define a function that converts a widget to a fudget. This mapping suffers from the same restrictions as the mapping from a fudget to a widget:

```
widgetToFudget :: Widget a -> F b c
widgetToFudget (Widget w) parent outputhandler
   = do w parent
        return nullHandler
```

The fudget takes the widget as its first argument, with the second and third arguments specifying the window the fudget will be realized in, and the output handler the fudget can use to send values on its output stream. The fudget is created by using the widget's realization function. The return value of the widget realization function is ignored, and the `nullHandler` function is used for the fudgets input handler. The wrapped widget cannot send output on the fudgets output stream, or read from the fudgets input stream, and so cannot interact with other fudgets.

Chapter 6

Conclusions and Future Work

There are a number of existing systems for the development of GUIs in a non-strict functional language, such as Fudgets, Gadgets and Haggis. However, the lower level components of these systems all solve very similar problems, such as how to handle I/O in functional languages, and also how to provide a structured interface to the event-driven model of windowing systems. Noble and Runciman [15] compare and contrast two systems for developing GUI's in functional languages. However, these systems are seperate standalone entities. A number of low level interfaces for windowing systems have recently been developed, such as Finne's X-Library bindings [6] which are part of the Haggis system. Reid [20] has also developed an interface for both the X-windows system and the Windows 95/NT systems as an extension to the Hugs functional programming system. This system has been used as the basis for an active virtual reality markup language (active VRML) implementation, making use of a graphics library [7], and a cooperative version of Concurrent Haskell [8].

We have presented the details of a system for the construction and comparison of graphical user interfaces in a purely functional language. The system is intended as a framework for the research and development of high-level abstractions for constructing graphical user interfaces. Illustrating this we have described two high-level abstractions that are built on the framework. The first of these abstractions, Widgets, uses a number of ideas from the Haggis and TK-Gofer systems, but does not make use of concurrency as Haggis does. The second, Fudgets, is implemented using an alternative representation to the original stream processing model used by Hallgren and Carlsson. The alternative representation is not however expressive enough to model the looping combinators of the original Fudgets system. This is a result that has not previously been noted.

The Widget and Fudget systems differ in the approach they take to structuring the GUI component of an application at a high-level. In particular, they differ in the data structures offered to the programmer for representing GUI components and also the way in which these components can be combined and physically laid out on the display screen. The Embracing Windows framework provides a basis for building and comparing systems such as the Widget and Fudget systems.

One avenue of research that could be particularly interesting is to try and use the framework to explore relationships between high-level abstractions like Fudgets and Widgets, such as their relative expressive power. Chapter 5 presented a restricted mapping between the two systems. Preliminary work has already revealed that this mapping can be strengthed to a complete mapping from fudgets to widgets. The formalization of the relationships between systems such as Fudgets and Widgets may benefit from a layered approach; by specifying

a formal semantics for the Embracing Windows system, the semantics of these high-level systems could be simplified.

There are also a number of other high-level abstractions for creating GUIs in functional languages that could be implemented using this framework, such as one based on the Concurrent Clean I/O system. The Clean language uses a type system incorporating uniqueness types [22]. A type can be annotated to be unique, indicating that a value of this type must not be shared at the point in an evaluation where the value is required. It is interesting to note that the monadic style of I/O can be encoded using uniqueness types. The paradigm used by the Clean system for constructing graphical interactive programs does not inherently rely upon the uniqueness type system for performing I/O operations. As such, the paradigm could be implemented using a different low level I/O mechanism such as monadic I/O. We have performed some preliminary work attempting to construct a system based on the Embracing Windows framework that uses the Clean paradigm for constructing graphical interactive programs. The results of this work indicate that there are no inherent difficulties in building such a system.

Currently, the Widgets and Fudgets systems both support a form of automatic layout to aid the GUI programmer. The layout system is very similar in both of these systems. A logical step would be to try and abstract the layout system away from the details of the particular systems. Pragmatically, the layout of graphical components is often performed using a GUI-builder. GUI-builders make use of direct manipulation to allow the user to literally draw the interface they want. Such graphical tools are important in the development of graphical interactive applications. However GUI-builders cannot cope with all possible layouts, and have great difficulty with layouts that can change dynamically. Having a programmatic mechanism for specifying the layout of components is therefore still important. In general we would expect most application interfaces to be handled by a GUI-builder, and only rarely would the full expressiveness of the programmatic mechanism be required. The output of such GUI-builders is often in terms of the programmatic mechanism.

When developing graphical applications the development environment can play a significant role in making the process quick and easy. A GUI-builder is just one example of a tool that would form part of a development enviornment for graphical applications. Abandoning the concept of flat text files for specifying the entire code of a graphical application can lead to a development environment where the programmer literally draws the required interface, and then proceeds to write small blocks of code attaching them to the components of the interface. A good example of this concept in action is seen in the Visual Basic[1] programming system, where programs are built by drawing forms and then writing a number of small blocks of code. A specific block of code describes the behaviour of a specific component on a form. However, Visual Basic is based on an imperative language and whether the approach will work well for declarative languages is a topic for further research.

The Embracing Windows framework has benefited from being implemented in an interpreter. Since design of graphical interfaces is very much an evolutionary process, it is advantageous to have a quick turn around from modifying the code specifying the interface to seeing the interface realized on a computer screen. Once the interface has been evolved however, then a compiler is a better choice of tool to give an efficient application.

Finally, in order to shoehorn systems such as Haggis, and Gadget Gofer into the Embracing Windows framework it appears that a form of concurrency will be necessary. This could be added into the framework as a new layer that provides minimal support for concurrency upon

[1] Visual Basic is a registered trademark of the Microsoft Corporation

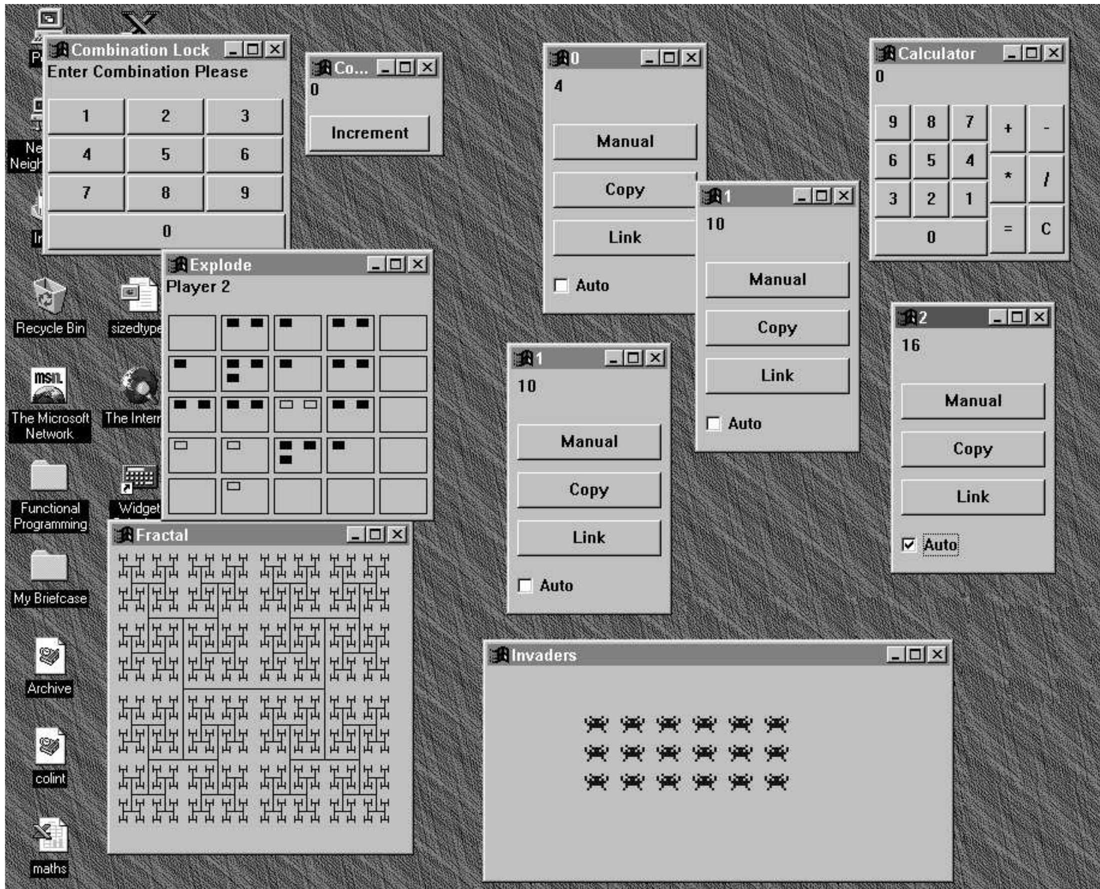


Figure 6.1: A Potpourri of graphical applications

which the particular communication protocols could be built. The form of concurrency that best suits graphical user interfaces is still an open question.

A selection of graphical applications that have been written using the Embracing Windows framework can be seen in Figure 6.1.

6.1 Acknowledgements

I would like to especially thank Mark P. Jones for his many valuable comments, suggestions and careful reviews of draft versions of this report. Also thanks to Graham Hutton and Benedict R. Gaster for numerous useful discussions and comments which, I hope, have enabled me to improve the content and presentation of this work. Also thanks to Rob Noble for some insightful comments regarding the calculator example presented in the appendices. This work was carried out while the author was a member of the Functional Programming Group at the University of Nottingham, UK, with financial support from the University of Nottingham.

Appendix A

Example Applications

This appendix presents various applications that have been written in both the Widgets and Fudgets systems. The main goal when developing these applications was their reusability. This is illustrated particularly well with the combination lock example in Appendix A.2, which reuses the entire numeric keypad developed for the calculator in Appendix A.1.

A.1 A Calculator

A common test of GUI development systems is to write an application modelling a simple desk calculator. The application should present an interface to the user very similar to that of a real calculator, with a display screen and a number of buttons for entry of numbers, and basic operations. The widget version of the calculator is shown in Figure A.1.

A.1.1 The Calculator State Machine

Before describing the implementation of the GUI for the calculator, we present the underlying model of the calculator itself. This model corresponds to the application part of the calculator program, as opposed to the GUI part.

The behaviour of the calculator can be described by a simple finite state machine. The input to the calculator is just a character corresponding to the key pressed by the user. The

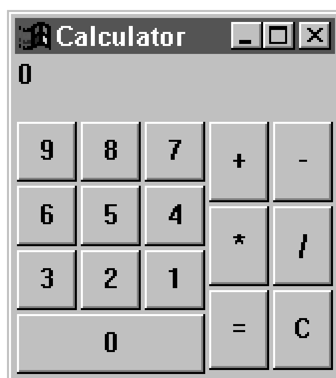


Figure A.1: A Widget calculator

output of the calculator is modelled as a pair of values, the first of which corresponds to the value on the calculator's display. The second value of this pair is a function, modelling all the calculations since the equal key was last pressed. A simple finite state machine such as this can be easily implemented in a functional language:

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

eval                    :: Char -> CalcState -> CalcState
eval c state | isDigit c = evalDigit (ord c - ord '0') state
             | otherwise = evalOperation c state

evalDigit          :: Int -> CalcState -> CalcState
evalDigit n (d, a) = (10 * d + n, a)

evalOperation           :: Char -> CalcState -> CalcState
evalOperation 'C' (d, a) = (0, id)
evalOperation '=' (d, a) = (a d, const (a d))
evalOperation op  (d, a) = (0, evalOperator op (a d))

evalOperator    :: Char -> (Int -> Int -> Int)
evalOperator '+' = (+)
evalOperator '-' = (-)
evalOperator '*' = (*)
evalOperator '/' = (div)
```

The `eval` function describes the semantics of pressing a key on the calculator's keypad. If the clear, "C", key is pressed then the display is zeroed, and the accumulating function is set to the identity function. If the equal key is pressed, then the display is updated by applying the accumulating function to the current display value. The accumulating function becomes a constant function returning this same value. When a number key is pressed then the value being displayed is updated, multiplying it by ten and adding the number corresponding to the key pressed. The accumulating function does not change in this case. The only remaining keys correspond to the arithmetic operators, and pressing one of these sets the display to zero, and partially applies the appropriate arithmetic operator to the result of applying the accumulating function to the current display value.

A driver program is required to make the calculator useful, for instance a simple text based calculator can be implemented as:

```
main :: IO ()
main  = calc (0, id)

calc       :: (Int, Int -> Int) -> IO ()
calc (d, a) = do putStr ("\nDisplay: " ++ show d ++ "\n")
                 c <- getChar
                 calc (eval c (d, a))
```

The semantics described in this section associates the same precedence level to all operators. However, experimenting with real desktop calculators results in a suprising array of differing behaviours regarding precedence of arithmetic operators. The problem of having the

same precedence level for all operators is not solved here as we are mainly concerned with the interface for the calculator rather than the semantics of the calculator.

A.1.2 A Widget Graphical User Interface

The GUI of the calculator is modelled as a widget that appears in a top level window. However, note that because the entire calculator is encapsulated as a Widget, it does not have to be used in a top level window, but could be used to build more complicated Widgets. For example, a simple application that composes a number of calculators next to each other in a line can be easily written using the `hbox` combinator:

```
main :: IO ()
main  = startProg w
        where w = do wopen "Multiple Calculators" multicalcW
                     return ()

multicalcW :: Widget ()
multicalcW  = hBox (do calcW
                       calcW
                       calcW)
```

The `calcW` function is the calculator widget that we describe in the remainder of this section.

The calculator maintains some state storing the current value of the display, and an accumulator function, indicating the calculation entered so far. This is just the state as described in Appendix A.1.1:

```
type CalcStVar = Ref CalcState
```

The basic calculator widget creates an initial value of the state, and the widgets for the display and calculator keypad. These widgets are composed vertically, so that the display appears above the keypad. The state and a handle for the display are passed to the keypad widget so that the state can be modified and the display updated when keys are pressed:

```
calcW :: Widget ()
calcW  = vbox (do st <- stateW (0, id)
                  display <- dispW
                  keysW st display)
```

The display is implemented using the standard `textW` widget. However, a thin margin is wrapped around this widget to make the visual appearance better.

```
dispW :: Widget TextLabel
dispW  = margin 1 (textW "0" display_size)

display_size = (100, 30) :: Size
```

A generic keypad widget is constructed using the `matrix` layout combinator which takes a width of the matrix, and a list of widgets to be formed into a matrix shape. The widgets are used to build up the lines of the matrix, so that each line has at most the specified number of widgets on it. The `keypadW` function creates a keypad widget given a list of pairs. The pairs define the text label and behaviour of each key in the keypad:

```
keypadW           :: Int -> [(String, GUI ())] -> Widget ()
keypadW width keys = do matrix width (map key keys)
                        return ()
```

The keys themselves are generated using the `key` function, which adds a thin margin around the buttons for a better visual appearance:

```
key                :: (String, GUI ()) -> Widget PushButton
key (label, action) = margin 1 (buttonW label button_size action)

button_size = (30, 30) :: Size
```

The keypad of the calculator consists of two separate keypads, one for the number keys, and one for the operation keys. A generic number keypad is implemented in terms of the `keypadW` widget. The `numberPadW` function takes a function as its argument that describes the behaviour of the keypad. This function expects an integer value corresponding to a key that has been pressed on the keypad as its first argument:

```
numberPadW  :: (Int -> GUI ()) -> Widget ()
numberPadW f = keypadW 3 (map formkey [9,8..0])
               where formkey n = (show n, f n)
```

The number keypad for the calculator is implemented in terms of the generic `numberPadW` combinator. The function specifying the behaviour of the keys in the keypad uses the function `keyOp` to modify the state of the calculator and update the display. The state modifying function used in this case is the `evalDigit` function determining the semantics of entering digits into the calculator.

```
numbersW        :: CalcStVar -> TextLabel -> Widget ()
numbersW st disp = numberPadW (\n -> keyOp st disp (evalDigit n))

keyOp          :: CalcStVar -> TextLabel -> (CalcState -> CalcState) -> GUI ()
keyOp st disp f = do (d, a) <- getRef st
                     let (d', a') = f (d, a)
                     setRef st (d', a')
                     setText disp (show d')
```

The keypad for the operations uses the `keyPadW` combinator and the `keyOp` function in a similar way to the `numbersW` function. In this case however, the state modifying function is `evalOperation`, determining the semantics of entering an operation into the calculator.

```
operationsW        :: CalcStVar -> TextLabel -> Widget ()
operationsW st disp = keypadW 2 [([op], keyOp st disp (evalOperation op))
                                          | op <- "+-*/=C"]
```

Both of the keypads are combined to form the single keypad used in the calculator. The keypads are placed next to each other horizontally using the `hbox` layout combinator:

```
keysW        :: CalcStVar -> TextLabel -> Widget ()
keysW st disp = hbox (do numbersW st disp
                         operationsW st disp)
```

Finally, the driver program for the calculator realizes the calculator widget in a top level window:

```
main :: IO ()
main  = startProg w where
        w = do wopen "Calculator" calcW
               return ()
```

A.1.3 A Fudget Graphical User Interface

The fudgets calculator is similar to the widgets version, and starts by realizing the calculator fudget in a top level window. Again the entire calculator is encapsulated as a single fudget, and so may be used to build more complicated fudgets:

```
main :: IO ()
main  = fudlogue (shellF "Calculator" calcF)
```

The state is the same as for the Widget calculator, consisting of the current value on the display of the calculator, and an accumulator storing the current computation in progress. Ideally, we would like to use a recursive stream processor to model the state, but due to the problem described in Section 4.4 this will not work. Instead we use mutable state indirectly by using the `mapAccumlSP` function.

The main calculator consists of four components, a display screen fudget, two abstract fudgets, and a fudget representing the keypad of the calculator. The two abstract fudgets encapsulate the logic of the calculator, while the remaining fudgets describe its user interface. These four fudgets are combined together serially, and the physical layout of the fudgets is specified using the `vBoxF` layout function:

```
calcF :: F Click a
calcF  = vBoxF (displayF                >==<
                absF displayState       >==<
                absF (stateSP (0, id))  >==<
                keysF)
```

The display screen is based on the standard integer display fudget, `intDispF`, but is wrapped by the `marginF` fudget combinator that adds a margin around the top, bottom, left and right of a fudget. The first parameter to the `marginF` combinator indicates the size of this margin.

```
displayF :: F Int a
displayF  = marginF 1 (intDispF display_size)

display_size = (80, 30) :: Size
```

The current value to be displayed on the calculator's screen must be extracted from the state of the calculator, and this is the job of the `displayState` abstract fudget. It is a stream processor that reads in a value indicating the current state of the calculator, and discards the accumulator component, writing the current display value to its output stream.

```
displayState :: SP CalcState Int
displayState  = getSP (\(d, a) -> putSP d displayState)
```

In the same manner that we abstracted the notions of a generic keypad, and of a numeric keypad from the widget version of the calculator application we do the same for the fudget version.

Starting with the generic notion of a keypad, we build a fudget that uses the `matrixF` layout combinator to build a keypad. The `keypadF` function takes the width of the keypad and a list specifying the details of the keys. Each key is determined by its label and a value that is emitted from the keypad when it is pressed. Just as we added a thin margin to the keys in the keypad for the widget version we do the same here. An abstract fudget is used to produce the required value when a particular button representing a key in the keypad is pressed:

```
keypadF           :: Int -> [(String, a)] -> F Click a
keypadF width keys = matrixF width (>*<) (map key keys)

key                :: (String, a) -> F Click a
key (label, action) = const action >^=< marginF 1 (buttonF label button_size)

button_size = (30, 30) :: Size
```

A numeric keypad fudget, `numberPadF`, can be built using the generic keypad combinator `keypadF` in a similar way to the `numberPadW` widget. Instead of taking a function describing the action to be performed when one of the digits on the numeric pad is pressed, the number pressed is emitted on the output stream of the fudget:

```
numberPadF :: F Click Int
numberPadF  = keypadF 3 (map formkey [9,8..0])
              where formkey n = (show n, n)
```

The calculator's keypad is composed of two separate components, the first is a numeric keypad, whilst the second is a keypad whose keys correspond to the available arithmetic operations. The numeric keypad is defined as a specialised version of the more generic `numberPadF` fudget, by simply processing the ouput digit with the appropriate function of the state machine. The arithmetic operation keypad is built using the `keypadF` fudget by specifying the output values for the keys to be the application of the `evalOperation` function from the state machine to the character representing the particular operation:

```
numbersF :: F Click (CalcState -> CalcState)
numbersF  = absF (mapSP evalDigit) >==< numberPadF

operationsF :: F Click (CalcState -> CalcState)
operationsF  = keypadF 2 [([op], evalOperation op) | op <- "+-*/=C"]
```

The entire calculator keypad is formed by combining the specialised numeric keypad and the arithmetic operation keypad in parallel:

```
keysF :: F Click (CalcState -> CalcState)
keysF  = hBoxF (numbersF >*< operationsF)
```

The internal state of the calculator is maintained by the abstract fudget `stateSP`, which accepts state modifying functions as its input and outputs the new values of the state under these functions. The state is implemented using the standard `mapAccumlSP` function which

requires a function that will specify how the state is to be modified and what the output value will be. In this case the output value is the same as the modified state hence in the tuple returned by `modify` the two components are the same.

```
stateSP       :: CalcState -> SP (CalcState -> CalcState) CalcState
stateSP state = mapAccumlSP modify state
                where modify st f = let st' = f st
                                    in (st', st')
```

The semantics of the calculator logic is specified by the `evalOperation` and `evalDigit` functions just as it was for the widgets version, see Section A.1.1.

A.1.4 Comparison of Widget and Fudget GUIs

The two interfaces developed in the preceding sections look identical on screen. The code is also remarkably similar, with almost all of the widget functions having obvious corresponding functions in the fudget version. The stream processors, `stateSP` and `displayState`, correspond to the `keyOp` function of the widget version. However, one important difference is that in the widget version the abstractions for keypads require the actions that are to be performed, when the keys are pressed, as arguments. In the fudget version these actions can be determined at a later point as the keypads simply emit a value indicating which key has been pressed.

A.2 A Combination Lock

At the Glasgow GUIFest in 1995, one of the suggestions for applications to test GUI development systems was a combination lock. The idea is to build a graphical component modelling a combination lock that can be used wherever a button component could be used.

A.2.1 A Widget Combination Lock

The combination lock we model here is based on a digital combination lock with a keypad for entering combinations. We will consider only numeric combinations here, but it is easy to extend the code to handle combinations using other symbols.

Firstly, we will need a graphical component modelling a keypad. Fortunately, we have already described a numeric keypad in the implementation of the calculator widget (Section A.1.2), and we can reuse the `numberPadW` function for the combination lock.

The combination lock needs to store the current code entered as part of its state. The behaviour exhibited when the correct code is entered can be changed by the user of the combination lock widget, and so must also be stored as part of its state:

```
type Code      = [Int]
type CombState = (Code, GUI ())
type CombStVar = Ref CombState
```

Codes are represented as lists of integer digits. The first component in the `CombState` type represents the code that the user has entered. The second component in the `CombState` type represents the behaviour of the combination lock when the correct code has been entered.

The combination lock widget must be able to be used wherever a regular button widget could be used. All button widgets must be instances of the `Editable` type class, and so we

must make the combination lock widget an instance too. A data type declaration is required so that we can represent combination locks in a form that we can use to make an instance of the `Editable` type class:

```
data CombinationLock = CombinationLock CombStVar
```

This representation allows us to manipulate the combination lock through its state. We can make `CombinationLock` an instance of the `Editable` type class, but before we do this, we must make it an instance of the `Control` type class, as this is a subclass of the `Editable` type class:

```
instance Control CombinationLock where
   setText (CombinationLock st) text = return ()
   getText (CombinationLock st)      = return ""
```

Here for simplicity, we do not make use of the associated text for a combination lock control and implement dummy behaviour for altering or retrieving this text. This is expressed by the implementations of the `setText` and `getText` functions that have no side effects and return dummy values. Now we can define an instance of the `CombinationLock` data type for the `Button` type class:

```
instance Editable CombinationLock where
   onChange (CombinationLock st) handler'
      = do (current, handler) <- getRef st
           setRef st (current, handler')
   onCommit (CombinationLock st) handler = return ()
```

The implementation of the `onChange` function simply replaces the current handler used by the combination lock when the correct code is entered.

The actual combination lock is described by a widget combinator, `combLockW`, which creates a widget that, when realized in a window, will create an initial value of the combination lock's state, realize the keypad and return a value that can be used to further manipulate the combination lock:

```
combLockW              :: Code -> GUI () -> Widget CombinationLock
combLockW code handler = do st <- stateW ([], handler)
                            numberPadW (match st (reverse code))
                            return (CombinationLock st)
```

The master code that is passed to the `match` function is reversed as this simplifies the code for this function.

The logic behind the combination lock is all described by the `match` function that expresses the behaviour exhibited by the keypad when one of the buttons in the keypad is pressed. This function must construct the new code entered by the user by adding the new digit corresponding to the button pressed to the existing code. It must also compare the new code entered to the master code, if the two match then the behaviour specified by the second component of the state, the handler, must be performed. If the codes do not match, then we simply store the new code entered by the user in the state. However, the code entered by the user must never exceed the length of the master code, and so the new code entered by the user is truncated to the length of the master code:

Figure A.2: A counter using a combination lock

```
match             :: CombStVar -> Code -> Int -> GUI ()
match st master x = do (current, handler) <- getRef st
                       let current' = take (length master) (x : current)
                       if current' == master then
                          do handler
                             setRef st ([], handler)
                        else
                          setRef st (current', handler)
```

If the code entered by the user does match the master code, then the currently entered code is reset to an empty list, representing an empty code, so that the combination lock is reset, ready to receive another combination attempt.

It is quite simple to modify the counter application from Figure 3.2 to use a combination lock instead of a regular push button. The only function that needs to be changed is the `counter` function:

```
counter :: Int -> Widget CombinationLock
counter init
  = vbox (do (display, update) <- stateTextW display_size 0
             combLockW [1,2,3] (update (+1)))
```

The master code is chosen to be 123, and when this code is entered a function to add one to its argument is sent to the Widget created by the `stateTextW` combinator. The resulting application can be seen in Figure A.2.

Appendix B

Source Code

This section provides the Haskell source code for the Embracing Windows framework. The framework also requires a modified version of the Hugs functional programming system that implements the monadic primitives for windowing operations. These primitives are implemented in C, in a similar way to the primitives already built-in to Hugs for teletype I/O. The source code for these primitives is not included here.

B.1 I/O Primitives and Library Files

B.1.1 Types

This module contains the type definitions for particular types used throughout the Embracing Windows framework. It also includes data type definitions for built-in types corresponding to window handles, device contexts, and objects. Objects are used for drawing graphics; currently, the only type of object available is a pen.

```
> module Types
> where
```

Pairs of integers are used very frequently, and so we define a type for them:

```
> type Vector   = (Int, Int)
```

Vectors are used in many different ways to represent points, sizes and rectangles. We define type synonyms and useful access functions for each of these cases to make the code easier to comprehend. Using data type definitions would be better as we gain stricter type checking and could also make use of the Haskell 1.3 record syntax which gives us the selector functions automatically:

```
> type Point    = Vector
> type Size     = Vector
> type Rect     = (Vector, Vector)

> getX :: Size -> Int
> getX (x, _) = x

> getY :: Size -> Int
> getY (_, y) = y

> getSize :: Rect -> Size
```

```
> getSize (_, size) = size

> getPoint :: Rect -> Point
> getPoint (point, _) = point

> type Colour = Int
```

The following data types are built-in to the modified verision of Hugs, and are used to represent windows, device contexts, and objects respectively. An null value for the `Window` data type is defined, which is `nullWindow` useful for cases when a value of type `Window` is required but not really important. Equality is defined on values of type `Window`, as this is useful in determining if an event is related to a specific window.

```
> data Window
> data DC
> data Object

> primitive nullWindow    "primNullWindow" :: Window
> primitive primEqWindow                   :: Window -> Window -> Bool
> instance Eq Window where
>     (==) = primEqWindow
```

An event in Windows 95 is comprised of a window, message identifier, and two extra integer parameters for event specific information. We model this as a 4-tuple:

```
> type Event = (Window, Int, Int, Int)
```

B.1.2 Monadic Primitives

This module contains the definitions of the primitives required for the windows interface. There are three classes of primitives, event loop primitives, window primitives, and graphic primitives.

```
> module Windows_API
> where

> import Types
```

The basic event loop primitives include primitives for starting an event loop, passing an event to the default handler, and for quiting an event loop:

```
> primitive primEventLoop      :: (Event -> IO Int) -> IO ()
> primitive primDefaultHandler :: Event -> IO Int
> primitive primQuitEventLoop  :: IO ()
```

The windowing primitives provide functions for creating and destroying windows, setting and retrieving the caption of a window, setting and retrieving the size of a window, retrieving a list of the open windows, and setting the visibility of a window. The window creation function takes a string value as its first argument that determines the window class to use. A window class can be used in Windows 95 to specify the initial parameters of a window. A number of window classes are predefined in Windows 95, such as "button", "edit", "static", corresponding to button controls, editable text field controls, and static text labels. The primitive for obtaining a list of open windows is used to provide a default behaviour for exiting from an application, by automatically closing any open windows. The first argument to the `primShowWindow` primitive is used to specify whether a window should be made visible or invisible. The value `True` indicates that the window should be made visible, the value `False` indicates that the window should be made invisible.

```
> primitive primCreateWindow  :: String -> Window -> Bool -> IO Window
> primitive primDestroyWindow :: Window -> IO ()

> primitive primSetWindowText :: Window -> String -> IO ()
> primitive primGetWindowText :: Window -> IO String

> primitive primSetWindowRect :: Window -> Rect -> IO ()
> primitive primGetWindowRect :: Window -> IO Rect

> primitive primGetWindows    :: IO [Window]
> primitive primShowWindow    :: Bool -> Window -> IO ()
```

Two primitives are provided for applications that require some notion of timing to operate. The first primitive sets a *timer*, that will cause a special event corresponding to a tick to occur at specific intervals of time. The second primitive stops such ticks from occuring by removing the *timer*:

```
> primitive primSetTimer      :: Window -> Int -> Int -> IO ()
> primitive primKillTimer     :: Window -> Int -> IO ()
```

The following primitives are useful for drawing graphics. The first two are used to obtain and release device contexts needed for drawing. Only two basic drawing primitives are supported, for moving to and drawing a line to a particular point in a window. There are many other primitives that could have been included, such as ones for drawing circles or polygons, but we omit these for reasons of simplicity, and also most of these more complicated primitives can be encoded using the basic ones provided here. Support for drawing text is provided by the `primDisplayText` primitive. Objects are used to alter the device context, the only one currently provided is the pen. Primitives are specified for selecting an object into a device context, and for deleting an object from memory. A simple pen object creation primitive is also supplied. Finally the `primBeginPaint` and `primEndPaint` primitives are supported to obtain and release a device context for drawing in a window in response to a paint message:

```
> primitive primGetDC         :: Window -> IO DC
> primitive primReleaseDC     :: Window -> DC -> IO ()

> primitive primMoveTo        :: DC -> Int -> Int -> IO ()
> primitive primLineTo        :: DC -> Int -> Int -> IO ()

> primitive primDisplayText   :: DC -> Int -> Int -> String -> IO ()

> primitive primSelectObject  :: DC -> Object -> IO Object
> primitive primDeleteObject  :: Object -> IO Bool

> primitive primCreatePen     :: Int -> Colour -> IO Object

> primitive primBeginPaint    :: Window -> IO DC
> primitive primEndPaint      :: Window -> IO ()
```

B.1.3 Window Constants

This module contains definitions for constants that are specific to the Windows 95 system.

```
> module Window_Constants
> where
```

The following constants determine the extra space that window borders and menus occupy in addition to the main client window area:

```
> window_extra_x = 8 :: Int
> window_extra_y = 27 :: Int
```

B.1.4 Table

The state maintained in the GUI monad is stored in tables. This module provides a simple implementation of tables.

```
> module Table
> where
```

A table is represented as an association list:

```
> type Table k e = [(k, e)]
```

A null table is representated as an empty association list:

```
> nullTable :: Table k e
> nullTable = []
```

The following functions are useful auxillary functions for comparing keys and retrieving the value from an entry in an association list:

```
> keyMatches :: Eq k => k -> (k, e) -> Bool
> keyMatches k (k', _) = k == k'

> getEntry :: (k, e) -> e
> getEntry (_, e) = e
```

The rest of the functions in this module provide the main interface to tables. The function `lookup` to retrieve the value corresponding to a particular key is missing here, as this function is predefined by the Haskell 1.3 prelude file. The `update` function can be used to update the contents of a table. The `addEntry` and `removeEntry` functions can be used to add and remove associations from a table:

```
> update :: Eq k => k -> (e -> e) -> Table k e -> Table k e
> update k f [] = []
> update k f (t:ts) | keyMatches k t = (k, f (getEntry t)) : ts
>                   | otherwise      = t : update k f ts

> addEntry :: Eq k => k -> e -> Table k e -> Table k e
> addEntry k e t = (k, e) : (removeEntry k t)

> removeEntry :: Eq k => k -> Table k e -> Table k e
> removeEntry k [] = []
> removeEntry k ((k', e'):es) | k' == k   = removeEntry k es
>                             | otherwise = (k', e') : removeEntry k es
```

B.1.5 Event loop primitives

This module declares a type class that allows us to refer to the monadic primitives for event loops by the same name regardless of the monad that the primitives have been lifted into. An instance of the type class is provided for the IO monad, which allows us to use the monadic primitives as they are declared in the Windows_API module.

```
> module EventSys
> where

> import Types
> import Windows_API

> class Monad m => EventSystem m where
>     eventLoop      :: (Event -> IO Int) -> m ()
>     defaultHandler :: Event -> m Int
>     quitEventLoop  :: m ()

> instance EventSystem IO where
>     eventLoop      = primEventLoop
>     defaultHandler = primDefaultHandler
>     quitEventLoop  = primQuitEventLoop
```

B.1.6 Window system primitives

This module declares a type class that allows us to refer to the monadic primitives for basic window operations by the same name regardless of the monad that the primitives have been lifted into. An instance of the type class is provided for the `IO` monad, which allows us to use the monadic primitives as they are declared in the `Windows_API` module. A window system is assumed to be event based and so the context for the `WindowSystem` type class includes the `EventSystem` type class. The `WindowSystem` type class includes a method not mentioned in the paper, `createWindow`. This is a more primitive window creation function that `createShellWindow` and is useful for creating controls and child windows as well as shell windows.

```
> module WinSys
> where

> import Types
> import Windows_API
> import EventSys

> class EventSystem m => WindowSystem m where
>     createWindow      :: String -> Window -> Bool -> m Window
>     createShellWindow :: String -> m Window
>     destroyWindow     :: Window -> m ()
>     setWindowCaption  :: Window -> String -> m ()
>     getWindowCaption  :: Window -> m String
>     setWindowRect     :: Window -> Rect -> m ()
>     getWindowRect     :: Window -> m Rect
>     getWindows        :: m [Window]
>     showWindow        :: Bool -> Window -> m ()
>     setTimer          :: Window -> Int -> Int -> m ()
>     killTimer         :: Window -> Int -> m ()
>     getDC             :: Window -> m DC
>     releaseDC         :: Window -> DC -> m ()
>     beginPaint        :: Window -> m DC
>     endPaint          :: Window -> m ()

> instance WindowSystem IO where
>     createWindow      = primCreateWindow
>     createShellWindow = primCreateShellWindow
>     destroyWindow     = primDestroyWindow
>     setWindowCaption  = primSetWindowText
```

```
>    getWindowCaption  = primGetWindowText
>    setWindowRect     = primSetWindowRect
>    getWindowRect     = primGetWindowRect
>    getWindows        = primGetWindows
>    showWindow        = primShowWindow
>    setTimer          = primSetTimer
>    killTimer         = primKillTimer
>    getDC             = primGetDC
>    releaseDC         = primReleaseDC
>    beginPaint        = primBeginPaint
>    endPaint          = primEndPaint
```

The `createWindow` primitive is a very general window creation function, but it is quite cumbersome to use. A simpler function for creating shell windows can easily be defined as:

```
> primCreateShellWindow :: WindowSystem m => String -> m Window
> primCreateShellWindow title
>    = do window <- createWindow "HugsWindow" nullWindow True
>         setWindowCaption window title
>         return window
```

B.1.7 Mutable Variables

This module declares a type class encapsulating the operations that characterise mutable variables. This allows the names for these operations to be reused regardless of the monad we are working in. The built-in data type for mutable variables and their associated operations are also declared. These built-in operations are used to declare an instance of the `MutVars` type class, so that the operations may be used in the `IO` monad.

```
> module IORef
> where

> class MutVars m where
>    newRef :: a -> m (Ref a)
>    getRef :: Ref a -> m a
>    setRef :: Ref a -> a -> m ()

> data Ref a

> primitive primNewRef  "newRef" :: a -> IO (Ref a)
> primitive primGetRef  "getRef" :: Ref a -> IO a
> primitive primSetRef  "setRef" :: Ref a -> a -> IO ()

> instance MutVars IO where
>    newRef = primNewRef
>    getRef = primGetRef
>    setRef = primSetRef
```

Equality on mutable variables is declared as it is in the `IORef` module since this does not depend upon the monad in which the mutable variable operations are lifted into:

```
> primitive eqRef  :: Ref a -> Ref a -> Bool
> instance Eq (Ref a) where
>     (==) = eqRef
```

B.2 Event Handling

B.2.1 GUI

This module defines the `GUI` monad and associated functions. The state that the `GUI` monad maintains is stored in a table containing tables that can be used to map an event to the appropriate behaviour. This whole structure is stored in a mutable variable.

```
> module GUI
> where

> import Types
> import Table
> import MutVar
> import Message
```

Event handlers are functions that take an event as an argument and process it. The response to an event may well involve manipulation of windows, and the event handlers themselves, and so the result of an event handler is a value of the `GUI` monad.

```
> type EventHandler = Event -> GUI ()
```

The state maintained by the `GUI` monad is stored in a table of type `Window_Table` whose entries are tables themselves. The whole state is stored as a mutable variable.

```
> type EventHandlers = Table EventType EventHandler
> type Window_Table = Table Window EventHandlers
> type GUIState = Window_Table
> type GUIStateVar = Ref GUIState
```

The following functions are useful for accessing and changing the state maintained by the `GUI` monad.

```
> getWindowTable :: GUIState -> Window_Table
> getWindowTable = updateWindowTable id

> setWindowTable :: Window_Table -> GUIState -> GUIState
> setWindowTable window_table = updateWindowTable (\_ -> window_table)

> updateWindowTable :: (Window_Table -> Window_Table) -> GUIState -> GUIState
> updateWindowTable f window_table = f window_table
```

The `GUI` monad is a state reader monad. The state encapsulated by this monad is a mutable variable that stores the responses required for particular events. Because the monad is a state reader monad, the mutable variable itself cannot be changed, but the value it contains can be changed, and thus the monad can act like a normal state transformer monad.

```
> newtype GUI a = GUI (GUIStateVar -> IO a)

> instance Functor GUI where
>   map f (GUI g) = GUI (\st -> do a <- g st
>                                  return (f a))

> instance Monad GUI where
>   return x        = GUI (\st -> return x)
>   GUI g >>= f     = GUI (\st -> do a <- g st
>                                    let GUI h = f a
>                                    h st)
```

The getenvGUI function is useful for extracting the mutable variable containing the state encapsulated by the GUI monad.

```
> getenvGUI :: GUI GUIStateVar
> getenvGUI = GUI return
```

Operations of type IO a can be lifted into the GUI monad by simply ignoring the mutable variable containing the state encapsulated by the GUI monad.

```
> liftGUI :: IO a -> GUI a
> liftGUI f = GUI (\_ -> f)
```

Since the GUI monad uses mutable variables, it is useful to have versions of the operations on mutable variables that work in the GUI monad. This can be achieved by providing an instance of the MutVars type class for the GUI type, with the methods being implemented as lifted versions of the original mutable variable primitives.

```
> instance MutVars GUI where
>    newRef a   = liftGUI (newRef a)
>    getRef a   = liftGUI (getRef a)
>    setRef a x = liftGUI (setRef a x)
```

The following functions provide a means to access and modify the state encapsulated by the GUI monad. The caller of these functions needs no knowledge of the fact that the state is stored in a mutable variable.

```
> updateGUIState :: (GUIState -> GUIState) -> GUI GUIState
> updateGUIState f = do st <- getenvGUI
>                       v <- getRef st
>                       setRef st (f v)
>                       return v

> getGUIState :: GUI GUIState
> getGUIState = updateGUIState id

> setGUIState :: GUIState -> GUI ()
> setGUIState state = do updateGUIState (\_ -> state)
>                        return ()
```

To make the GUI monad useful, we have to define a function to turn a value of type GUI a into one of type IO a so that it can be executed in a program. This requires supplying an initial value for the state that the GUI monad encapsulates.

```
> startingWithGUI :: GUI a -> GUIStateVar -> IO a
> startingWithGUI (GUI f) r = f r
```

B.2.2 Event Handlers

This module defines a family of coercion functions that unpack information from the Event data type and pass it on to a function to process the event. The different types of events are identified by a magic number specific to the Microsoft Windows 95 system.

```
> module EventHandlers
> where

> import Types
> import Message
```

```
> import GUI
> import Graphics

> handleDestroy :: (Window -> GUI ()) -> Event -> GUI ()
> handleDestroy f (w, 2, _, _) = f w

> handlePaint :: (Window -> Draw ()) -> Event -> GUI ()
> handlePaint f (w, 15, _, _) = paintInWindow w (f w)
```

The event processing function that the `handleLButtonDown` coercion function takes as its first argument receives the window the mouse button was pressed in, a boolean to indicate whether the mouse button was single or double clicked, and a vector describing the position of the mouse cursor when the mouse button was pressed. In a similar way, the `handleLButtonUp` coercion function passes on the window the mouse cursor was clicked in, and its position to the event processing function.

```
> handleLButtonDown :: (Window -> Bool-> Point -> GUI ()) -> Event -> GUI ()
> handleLButtonDown f (w, 513, _, lparam) = f w False (getVector lparam)
> handleLButtonDown f (w, 515, _, lparam) = f w True (getVector lparam)

> handleLButtonUp :: (Window -> Point -> GUI ()) -> Event -> GUI ()
> handleLButtonUp f (w, 514, _, lparam) = f w (getVector lparam)
```

The event processing function used with the `handleCommand` coercion function expects four parameters, the associated window, an identifier, handle of control, and a notification code. The notification code indicates whether the event is a notification from a control, a menu selection, or from a keyboard accelerator for a menu. The identifier specifies which control or menu item is involved. The handle of the control specifies which control sent the notification if the event represents a control notification, otherwise this value is null. child control, The `handleCommand` coercion function

```
> handleCommand :: (Window -> Int -> Int -> Int -> GUI ()) -> Event -> GUI ()
> handleCommand f (w, 273, wp, lp) = f w (loword wp) lp (hiword wp)
```

The `handleKey` coercion function passes on parameters to its event processing function indicating the window involved, a code for the key pressed, a boolean representing if the key was pressed or released, and two integers describing the number of times the keystroke is repeated and the state of modifier keys such as shift, alt, and ctrl.

```
> handleKey :: (Window -> Int-> Bool -> Int -> Int -> GUI ()) -> Event -> GUI ()
> handleKey f (w, 256, wp, lp) = f w wp True (loword lp) (hiword lp)
> handleKey f (w, 257, wp, lp) = f w wp False (loword lp) (hiword lp)
```

The integer passed to the event processing function by the `handleQuit` coercion function indicates the exit code, describing whether the application is being quitted for normal or abnormal reasons.

```
> handleQuit :: (Window -> Int -> GUI ()) -> Event -> GUI ()
> handleQuit f (w, 18, wp, _) = f w wp
```

The integer `handleTimer` passes to its event processing function is an identifier for the particular timer involved.

```
> handleTimer :: (Window -> Int -> GUI ()) -> Event -> GUI ()
> handleTimer f (w, 275, wp, _) = f w wp
```

`handleSize` passes an integer representing the type of sizing operation, such as maximizing, minimizing, restoring, or normal window sizing, to the event processing function.

```
> handleSize :: (Window -> Int -> Size -> GUI ()) -> Event -> GUI ()
> handleSize f (w, 5, wp, lp) = f w wp (getVector lp)
```

B.2.3 Message

This module contains an event type for the Windows 95 system, and some auxillary functions that are useful for packing/unpacking values from an event.

```
> module Message
> where

> import Types
```

The `hiword` and `loword` functions extract the high word, and low word from a 32 bit value.

```
> hiword :: Int -> Int
> hiword w = w `div` 65536

> loword :: Int -> Int
> loword w = w `mod` 65536
```

Vectors are commonly packed into a 32 bit value, and require unpacking using the `hiword` and `loword` functions, `getVector` performs this unpacking.

```
> getVector :: Int -> Vector
> getVector lparam = (loword lparam, hiword lparam)
```

The `makeParam` function is useful for constructing a 32 bit value from two 16 bit words.

```
> makeParam :: Int -> Int -> Int
> makeParam hi lo = lo + (hi * 65536)
```

The event data type supports only a handful of the possible events for the Windows 95 system. The events supported are window destruction, window painting, mouse button clicks, commands (menu selections, and control notifications), key presses, quitting applications, timer events, and window sizing. The `getEventType` function extracts the type of event from a value of type `Event` by examining the second parameter, the message parameter.

```
> data EventType = Destroy
>                | Paint
>                | LButtonDown
>                | LButtonUp
>                | Command
>                | Key
>                | Quit
>                | Timer
>                | Size
>                | Unknown
>                deriving Eq

> getEventType :: Event -> EventType
> getEventType (_, msg, _, _)
>   = case msg of
>        2   -> Destroy
>        15  -> Paint
```

```
>         513 -> LButtonDown
>         514 -> LButtonUp
>         515 -> LButtonDown
>         273 -> Command
>         256 -> Key
>         257 -> Key
>         18  -> Quit
>         275 -> Timer
>         5   -> Size
>         _   -> Unknown
>
```

B.2.4 Lifted Functions

To use the event loop primitives and the basic window operation primitives in the GUI monad requires appropriate instance declarations for the EventSystem and WindowSystem type classes. The implementation of the methods of these type classes simply uses the lifting function for the GUI monad to lift the primitives from the IO monad to the GUI monad.

```
> module Lift
> where

> import WinSys
> import GUI

> instance EventSystem GUI where
>    eventLoop handler               = liftGUI (eventLoop handler)
>    defaultHandler event            = liftGUI (defaultHandler event)
>    quitEventLoop                   = liftGUI quitEventLoop

> instance WindowSystem GUI where
>    createWindow c parent border    = liftGUI (createWindow c parent border)
>    createShellWindow title         = liftGUI (createShellWindow title)
>    destroyWindow window            = liftGUI (destroyWindow window)
>    setWindowCaption window text    = liftGUI (setWindowCaption window text)
>    getWindowCaption window         = liftGUI (getWindowCaption window)
>    setWindowRect window layout     = liftGUI (setWindowRect window layout)
>    getWindowRect window            = liftGUI (getWindowRect window)
>    getWindows                      = liftGUI getWindows
>    showWindow state window         = liftGUI (showWindow state window)
>    setTimer window id time         = liftGUI (setTimer window id time)
>    killTimer window id             = liftGUI (killTimer window id)
>    getDC window                    = liftGUI (getDC window)
>    releaseDC window dc             = liftGUI (releaseDC window dc)
>    beginPaint window               = liftGUI (beginPaint window)
>    endPaint window                 = liftGUI (endPaint window)
```

B.3 The Core

B.3.1 Windows

This file is the main windows interface. It has functions for maintaining the state of a GUI, that is the event handlers that are in place to handle events.

```
> module Windows
> where

> import Types
> import Table
> import GUI
> import Message
> import Lift
> import WinSys
> import MutVar
```

The following functions are used to process events. When an event occurs the `mainHandler` function encapsulates the state of the application into the `GUI` monad, and passes the even to the `processEvent` function. This function looks up the event in the application's state, first looking for the table containing responses to events for the particular window that the event is associated with, and then using the type of the event to determine the response. If no entry is found, then the response is determined by the default event handler, otherwise the response retrieved from the state is performed. If the response is determined by the default event handler, then its return value must be passed back to the window system (this is an idiosyncrasy of the Windows 95 system).

```
> getEventHandler :: GUIState -> Window -> EventType -> Maybe EventHandler
> getEventHandler state window eventtype
>     = do let window_table = getWindowTable state
>          eventhandlers <- lookup window window_table
>          lookup eventtype eventhandlers

> runEventHandler :: Maybe EventHandler -> Event -> GUI Int
> runEventHandler Nothing              event = defaultHandler event
> runEventHandler (Just event_handler) event = do event_handler event
>                                                 return 0

> processEvent :: Event -> GUI Int
> processEvent event = do let etype = getEventType event
>                             (w, _, _, _) = event
>                         state <- getGUIState
>                         let event_handler = getEventHandler state w etype
>                         runEventHandler event_handler event

> mainHandler :: GUIStateVar -> Event -> IO Int
> mainHandler st ce = startingWithGUI (processEvent ce) st
```

The initial state for all graphical applications is an empty table, which indicates that no windows have been created, and that all events are to be processed using the default event handler.

```
> initGUIState :: GUIState
> initGUIState = nullTable
```

A graphical application is started using the `startProg` function, which creates the state to store the table used for determining responses to events. The application is then given an opportunity to create windows, menus, buttons, and other graphical components, installing the appropriate responses to events in the state. Finally the event loop is entered, and the application then becomes responsive to interaction from the user.

```
> startProg :: GUI () -> IO ()
> startProg w = do st <- newRef initGUIState
>                  startingWithGUI w st
>                  eventLoop (mainHandler st)
```

The basic window creation primitive can be used to create so called child windows, which have another window as their parent, and exist inside this other window. A useful abstraction for creating these windows, not only creates the window, but sets its position and size, and also makes it visible.

```
> mkChildWindow :: Window -> Rect -> GUI Window
> mkChildWindow parent layout
>     = do window <- createWindow "HugsWindow" parent False
>          setWindowRect window layout
>          showWindow True window
>          return window
```

When a new window is created, we add a new entry to the state; this is a table that will contain the mappings between events and responses for the new window. This table is initially empty indicating that all responses are to be determined by the default event handler built-in to the window system.

```
> addWindow :: Window -> GUI ()
> addWindow window
>     = do state <- getGUIState
>          let window_table = getWindowTable state
>              window_table' = addEntry window nullTable window_table
>              state' = setWindowTable window_table' state
>          case (lookup window window_table) of
>           Nothing -> setGUIState state'
>           Just _ -> return ()
```

The following functions provide a simple interface for altering the responses to events that are stored in the graphical application's state. The functions support removing existing responses for events, and adding new responses for events.

```
> addHandler :: Window -> EventType -> EventHandler -> GUI ()
> addHandler window event handler
>   = do addWindow window
>        updateGUIState f
>        return ()
>   where f = updateWindowTable (update window (addEntry event handler))

> removeHandler :: Window -> EventType -> GUI ()
> removeHandler window event
>   = do updateGUIState f
>        return ()
>   where f = updateWindowTable (update window (removeEntry event))
```

When an application wishes to shut itself down, it can call this function. Closing of open windows is automatically taken care of by obtaining a list of all the open windows belonging to

the application, and then destroying these windows. The state for the application is emptied before destroying the windows so that the application does not process any messages relating to the windows destruction.

```
> quitApp :: GUI ()
> quitApp  = do window_list <- getWindows
>               updateGUIState f
>               mapM_ destroyWindow window_list
>               quitEventLoop
>               return ()
>              where f state = setWindowTable nullTable state
```

B.3.2 On Handlers

This module provides a family of functions useful for defining the behaviour of a graphical application. The application's state is altered to set the behaviour in response to an event. Also, the information packed in the event structure is unpacked according to the type of the event, requiring the programmer to supply a function using the unpacked information.

```
> module OnHandlers
> where

> import Types
> import GUI
> import Windows
> import EventHandlers
> import Lift
> import Windows_API

> onDestroy :: Window -> (Window -> GUI ()) -> GUI ()
> onDestroy window handler = addHandler window Destroy (handleDestroy handler)

> onPaint :: Window -> (Window -> Draw ()) -> GUI ()
> onPaint window handler = addHandler window Paint (handlePaint handler)

> onLButtonDown :: Window -> (Window -> Bool -> Point -> GUI ()) -> GUI ()
> onLButtonDown window handler
>   = addHandler window LButtonDown (handleLButtonDown handler)

> onLButtonUp :: Window -> (Window -> Point -> GUI ()) -> GUI ()
> onLButtonUp window handler
>   = addHandler window LButtonUp (handleLButtonUp handler)

> onCommand :: Window -> (Window -> Int -> Int -> Int -> GUI ()) -> GUI ()
> onCommand window handler = addHandler window Command (handleCommand handler)

> onKey :: Window -> (Window -> Int -> Bool -> Int -> Int -> GUI ()) -> GUI ()
> onKey window handler = addHandler window Key (handleKey handler)

> onQuit :: Window -> (Window -> Int -> GUI ()) -> GUI ()
> onQuit window handler = addHandler window Quit (handleQuit handler)

> onTimer :: Window -> (Window -> Int -> GUI ()) -> GUI ()
> onTimer window handler = addHandler window Timer (handleTimer handler)

> onSize :: Window -> (Window -> Int -> Size -> GUI ()) -> GUI ()
> onSize window handler = addHandler window Size (handleSize handler)
```

A useful abstraction is to create a window with a particular title, and with a predefined response of shutting down the application when closed.

```
> mkWindow :: String -> GUI Window
> mkWindow text
>     = do window <- createShellWindow text
>          onDestroy window (\_ -> quitApp)
>          return window
```

B.3.3 Controls

This module provides support for the built-in controls of a windowing system. A hierarchy of type classes is used to capture the commonalities between different controls.

```
> module Controls
> where

> import Types
> import GUI
> import Lift
> import Windows
> import OnHandlers
```

A control has the concept of an associated piece of text, such as the label on a button, and also has a particular size. These characteristics are captured by the `Control` type class:

```
> class Control a where
>     setText :: a -> String -> GUI ()
>     getText :: a -> GUI String
>     setRect :: a -> Rect -> GUI ()
>     getRect :: a -> GUI Rect
```

An editable control supports the concepts of change and commit. Change is when the contents of an editable control changes as a result of interaction with the user. Commit is when the contents of an editable control is fixed at its current value in response to a user action. An example of an editable control is an editable text field. Whenever the text of the text field is edited, the behaviour specified by the `onChange` function is invoked. Commit occurs when the user presses the return key in the text field:

```
> class Control a => Editable a where
>     onChange :: a -> GUI () -> GUI ()
>     onCommit :: a -> GUI () -> GUI ()

> class Editable a => Button a where
>     setState :: a -> Bool -> GUI ()
>     getState :: a -> GUI Bool
```

A push button is implemented by wrapping the built-in control inside of a transparent window. Both of the windows involved, the one for the control itself, and the transparent window wrapping it and stored in the representation of a push button. This allows the button to be easily manipulated. The operation of altering the text of the control uses the control's own window, whereas altering the size of the control alters the size of both the control's window and the surrounding transparent window. The button's content is considered to have changed whenever it is pushed, and thus the behaviour specified by the `onChange` method is invoked whenever this is the case. The position of a child window is specified relatively to its containing parent window, thus the position of a wrapped control window is (0, 0).

```
> data PushButton = PushButton Window Window

> instance Control PushButton where
>     setText (PushButton _ button) text = setWindowCaption button text
>     getText (PushButton _ button) = getWindowCaption button
>     setRect (PushButton window button) (xy, wh)
>       = do setWindowRect window (xy, wh)
>            setWindowRect button ((0, 0), wh)
>     getRect (PushButton window _) = getWindowRect window

> instance Button PushButton where
>     setState (PushButton _ _) state = return ()
>     getState (PushButton _ _)       = return False

> instance Editable PushButton where
>     onChange (PushButton window _) handler
>         = onCommand window (\w id ctl code -> handler)
>     onCommit (PushButton window _) handler = return ()
```

An editable text field is implemented in a similar manner to a push button, except the required responses for change and commit in the control are stored in the representation of the editable text field. This is necessary as the event indicating either a change or a commit has the same type. When changing the behaviour for either a change or a commit, we must maintain the behaviour for the other event, which we can retrieve from the editable text field's representation:

```
> data EditField = EditField Window Window (GUI ()) (GUI ())
> instance Control EditField where
>     setText (EditField _ edit _ _) text = setWindowCaption edit text
>     getText (EditField _ edit _ _) = getWindowCaption edit
>     setRect (EditField window edit _ _) (xy, wh)
>         = do setWindowRect window (xy, wh)
>              setWindowRect edit ((0, 0), wh)
>     getRect (EditField window _ _ _) = getWindowRect window

> instance Editable EditField where
>     onChange (EditField window button _ oncommit) handler
>         = onCommand window (\w id ctl code -> notify code button)
>         where notify n edit = if (n == 768) then handler
>                               else if (n == 1792) then oncommit
>                               else return ()
>     onCommit (EditField window button onchange _) handler
>         = onCommand window (\w id ctl code -> notify code button)
>         where notify n edit = if (n == 768) then onchange
>                               else if (n == 1792) then handler
>                               else return ()
```

A text label need only be an instance of the `Control` type class as it has none of the characteristics of an editable control. As such a wrapping transparent window is not required:

```
> data TextLabel = TextLabel Window
> instance Control TextLabel where
>     setText (TextLabel window) text = setWindowCaption window text
>     getText (TextLabel window)      = getWindowCaption window
>     setRect (TextLabel window) rect = setWindowRect window rect
>     getRect (TextLabel window)      = getWindowRect window
```

When creating a control, we need to specify a string identifying the type of control to be created, and a parent window. The basic window creation primitive does not automatically set

the position and size of the control, or it's associated text. To provide this extra functionality, we define a function `mkControl`, that also makes the control visible as well as setting its position and size.

```
> mkControl :: String -> Window -> String -> Rect -> GUI Window
> mkControl clss parent text size = do window <- createWindow clss parent True
>                                      setWindowRect window size
>                                      setWindowCaption window text
>                                      showWindow True window
>                                      return window
```

The following functions are used to create instances of particular controls. For push buttons and editable text fields, a child window is used to wrap the control window. This makes the processing of notification events from the control window easier to handle. Text labels have no notification events and so are not created inside of a child window:

```
> mkPushButton :: Window -> String -> Rect -> GUI PushButton
> mkPushButton parent text (xy, wh)
>     = do window <- mkChildWindow parent (xy, wh)
>          btn <- mkControl "button" window text ((0, 0), wh)
>          return (PushButton window btn)

> mkEditField :: Window -> String -> Rect -> GUI EditField
> mkEditField parent text (xy, wh)
>     = do window <- mkChildWindow parent (xy, wh)
>          edit <- mkControl "edit" window text ((0, 0), wh)
>          return (EditField window edit (return ()) (return ()))

> mkTextLabel :: Window -> String -> Rect -> GUI TextLabel
> mkTextLabel parent text (xy, wh)
>     = do window <- mkControl "static" parent text (xy, wh)
>          return (TextLabel window)
```

B.3.4 Graphics

This module provides support for drawing graphics in windows. A monad is introduced that encapsulates the context required for drawing.

```
> module Graphics
> where

> import Types
> import WinSys
```

Drawing in a window requires a device context, which is obtained in one of two ways depending upon the situation. If the drawing is taking place in response to the special paint event, then the device context can be obtained by using the `beginPaint` function, and must be freed using the `endPaint` function. For all other cases, a device context can be obtained using the `getDC` function, and is freed using the `releaseDC` function. These functions are specific to the Windows 95 system. The following two functions encapsulate these two ways of obtaining a device context:

```
> drawInWindow :: Window -> Draw a -> GUI a
> drawInWindow w d = do dc <- getDC w
>                       result <- startingWithDraw dc d
>                       releaseDC w dc
```

```
>                       return result

> paintInWindow :: Window -> Draw a -> GUI a
> paintInWindow w d = do dc <- beginPaint w
>                        result <- startingWithDraw dc d
>                        endPaint w
>                        return result
```

The Draw monad encapsulates a device context such that the programmer does not have to thread it throughout the code for drawing graphics. This monad is a state reader monad:

```
> newtype Draw a = Draw (DC -> GUI a)

> instance Functor Draw where
>    map f (Draw g) = Draw (\dc -> map f (g dc))

> instance Monad Draw where
>    return x     = Draw (\dc -> return x)
>    Draw g >>= f = Draw (\dc -> do a <- g dc
>                                   let Draw h = f a
>                                   h dc)

> startingWithDraw :: DC -> Draw a -> GUI a
> startingWithDraw dc (Draw d) = d dc
```

Arbitrary operations in the GUI monad can be lifted into the Draw monad simply by ignoring the device context that is encapsulated by the Draw monad:

```
> liftDraw :: GUI a -> Draw a
> liftDraw a = Draw (\_ -> a)
```

All operations from the EventSystem,WindowSystem and MutVars type classes are lifted into the Draw monad, so that these operations are all available while drawing graphics:

```
> instance EventSystem Draw where
>    eventLoop handler            = liftDraw (eventLoop handler)
>    defaultHandler event         = liftDraw (defaultHandler event)
>    quitEventLoop                = liftDraw (quitEventLoop)

> instance WindowSystem Draw where
>    createWindow clss parent brdr = liftDraw (createWindow clss parent brdr)
>    destroyWindow window         = liftDraw (destroyWindow window)
>    setWindowCaption window text  = liftDraw (setWindowCaption window text)
>    getWindowCaption window      = liftDraw (getWindowCaption window)
>    setWindowRect window layout   = liftDraw (setWindowRect window layout)
>    getWindowRect window         = liftDraw (getWindowRect window)
>    getWindows                   = liftDraw getWindows
>    showWindow state window      = liftDraw (showWindow state window)
>    setTimer window id time      = liftDraw (setTimer window id time)
>    killTimer window id          = liftDraw (killTimer window id)
>    getDC window                 = liftDraw (getDC window)
>    releaseDC window dc          = liftDraw (releaseDC window dc)
>    beginPaint window            = liftDraw (beginPaint window)
>    endPaint window              = liftDraw (endPaint window)

> instance MutVars Draw where
>    newRef a   = liftDraw (newRef a)
>    getRef a   = liftDraw (getRef a)
>    setRef a x = liftDraw (setRef a x)
```

A drawing system is characterised by operations for drawing lines, and text. It also supports the creation of pens with different colours and widths. These ideas are encapsulated into the `DrawingSystem` type class:

```
> class WindowSystem m => DrawingSystem m where
>    lineTo       :: Point -> m ()
>    moveTo       :: Point -> m ()
>    selectObject :: Object -> m Object
>    deleteObject :: Object -> m Bool
>    drawText     :: Point -> String -> m ()
>    createPen    :: Int -> Colour -> m Object
```

The `Draw` monad is a drawing system, as can be expressed by making it an instance of the `DrawingSystem` monad. The device context encapsulated by the `Draw` monad is used as an argument to the primitive functions for drawing in windows. Since the primitive functions were not included in any of the type classes we have defined, they must be lifted into the `GUI` monad explicitly:

```
> instance DrawingSystem Draw where
>    lineTo (x, y)        = Draw (\dc -> liftGUI (primLineTo dc x y))
>    moveTo (x, y)        = Draw (\dc -> liftGUI (primMoveTo dc x y))
>    selectObject obj     = Draw (\dc -> liftGUI (primSelectObject dc obj))
>    deleteObject obj     = Draw (\dc -> liftGUI (primDeleteObject obj))
>    drawText (x, y) text = Draw (\dc -> liftGUI (primDisplayText dc x y text))
>    createPen w c        = Draw (\dc -> liftGUI (primCreatePen w c))

> mapPoint :: (Int -> Int) -> Point -> Point
> mapPoint f (x, y) = (f x, f y)
```

An individual point can be drawn by drawing a line one unit in length. Coloured points require the creation of an appropriate pen with the correct colour and width:

```
> drawPoint :: DrawingSystem m => Point -> m ()
> drawPoint p = drawLine p (mapPoint (+1) p)

> drawCPoint :: DrawingSystem m => Point -> Colour -> Int -> m ()
> drawCPoint p c w = do pen <- createPen w c
>                       oldpen <- selectObject pen
>                       drawPoint p
>                       selectObject oldpen
>                       deleteObject pen
>                       return ()
```

A line can be drawn by moving to one end of the line, and drawing to the other end:

```
> drawLine :: DrawingSystem m => Point -> Point -> m ()
> drawLine p q = do moveTo p
>                   lineTo q
```

A filled rectangle can be drawn by drawing a sequence of lines next to each other. An unfilled rectangle just requires the perimeter to be drawn:

```
> drawFilledRect :: DrawingSystem m => Rect -> m ()
> drawFilledRect ((x1, y1), (x2, y2))
>     = sequence [drawLine (x1, y) (x2, y) | y <- [y1..y2]]

> drawRect :: DrawingSystem m => Rect -> m ()
> drawRect ((x1, y1), (x2, y2)) =
```

```
>    do moveTo (x1, y1)
>       lineTo (x1, y2)
>       lineTo (x2, y2)
>       lineTo (x2, y1)
>       lineTo (x1, y1)
```

A polygon is a list a points that, when joined together, form the polygon. The last point in the list is joined to the first point to close the shape:

```
> type Polygon = [Point]

> drawPolygon :: DrawingSystem m => Polygon -> m ()
> drawPolygon [] = return ()
> drawPolygon (p:ps) = sequence (moveTo p : map lineTo (ps ++ [p]))
```

Colours are internally implemented as 32 bit values with the least significant 8 bits representing the amount of red, the next significant 8 bits representing the amount of green, and the next significant 8 bits representing the amount of blue. This function should really be a primitive as it assumes a particular internal represenation for the `Colour` type:

```
> mkColour :: Int -> Int -> Int -> Colour
> mkColour r g b = r + (g * 256) + (b * 65536)
```

B.3.5 Embracing Windows Framework

This module imports all of the source code that comprises the Embracing Windows framework.

```
> module EmbracingWindows
> where

> import Windows_Constants
> import Windows
> import OnHandlers
> import Controls
> import Graphics
```

B.4 Widgets

The widget system implemented in this section has been inspired by both TK-gofer and Haggis. The layout combinators and the use of a monadic representation for a graphical component in particular bear strong similarities to the ideas used in Haggis. The decision not to use concurrency and to keep the overall system relatively simple stem from the approach that the TK-gofer system takes to expressing GUIs in a functional language.

B.4.1 Widgets

This module defines the data structure used for representing widgets, and associated manipulation functions.

```
> module WidgetCore
> where

> import EmbracingWindows
```

A layout request is characterised by three values, a minimum size, a natural size, and an appropriate function for changing the size of the graphical component associated with the layout request:

```
> data LR = LR {minSize :: Size, natSize :: Size, sf :: Rect -> GUI ()}
```

A widget is represented by a function whose argument identifies the window the widget is to be created inside of. The result of the function is a value of type `GUI (a, [LR])`, indicating that the function can perform I/O and that the result of this I/O is a pair comprising the real return value of the widget, and a list of layout requests. A list of layout requests is returned in the pair, as the basic combinators for combining widgets do not alter the layout of the widgets and so must preserve the layout requests:

```
> newtype Widget a = Widget (Window -> GUI (a, [LR]))
```

Combinators for the layout of widgets essentially take a list of layout requests and combine them into a single layout request. A function that does this is referred to as a placer, and for widgets the type of such functions is:

```
> type WPlacer = [LR] -> LR
```

It is useful to define a null sizing function that does nothing, and also a null layout request which requests no screen space and uses a null sizing function:

```
> nullSF :: Rect -> GUI ()
> nullSF (_, _) = return ()

> nullLR :: [LR]
> nullLR = [LR (0, 0) (0, 0) nullSF]
```

Widgets are combined using the two standard monadic combinators, `thenW` and `returnW`. The layout requests of widgets are combined by simply concatenating the two lists comprising the layout requests to obtain the new layout request list. The `Widget` type is made an instance of the `Monad` type class, allowing the use of the built-in notation for defining monadic functions. A map function can be defined for widgets, allowing us to define `Widget` as an instance of the `Functor` type class:

```
> thenW :: Widget a -> (a -> Widget b) -> Widget b
> thenW m n
>     =  Widget (\win ->
>               do let Widget m' = m
>                  (r, lra) <- m' win
>                  let Widget n' = n r
>                  (v, lrb) <- n' win
>                  return (v, lra ++ lrb))

> returnW :: a -> Widget a
> returnW a = Widget (\win -> return (a, nullLR))

> instance Monad Widget where
>   return        = returnW
>   (>>=)         = thenW

> mapW :: (a -> b) -> Widget a -> Widget b
> mapW f w = do x <- w
>              return (f x)

> instance Functor Widget where
>   map = mapW
```

A stateful widget has no on screen appearance, but creates a piece of mutable state and returns a reference to it:

```
> stateW :: a -> Widget (Ref a)
> stateW init = Widget (\win -> do st <- newRef init
>                                  return (st, nullLR))
```

A widget is realised in a window using the `wopen` function. This function creates a new window, and realises the widget inside of it. The widget returns a list of layout requests that must be combined to obtain a single layout request. The requests are combined by taking the largest values for the minimum, and natural sizes in both the horizontal and vertical dimensions. This corresponds to a default layout combinator that places widgets in a pile on top of one another. The newly created window is sized to the natural size of the combined layout request increased by a small amount corresponding to the width and height of window borders and menus. The size of the window borders and menus is specific to Windows 95. The combined sizing function is called to ensure that the widget is sized correctly, since the combined layout request size may be different to the size the components of the widget have requested. When a sizing event occurs for the newly created window, the combined sizing function is used to resize the widget. Finally the newly created window must be made visible:

```
> wopen :: String -> Widget a -> GUI Window
> wopen text (Widget wid)
>    = do win <- mkWindow text
>         (_, lrs) <- wid win
>         let (LR _ (sx, sy) sf) = pileWP lrs
>         setWindowRect win ((0, 0), (sx + window_extra_x, sy + window_extra_y))
>         sf ((0, 0), (sx, sy))
>         onSize win (\w state xy -> sf ((0, 0), xy))
>         showWindow True win
>         return win
```

The following function defines a widget placer for placing widgets in a pile. The combined minimum and natural sizes are obtained by taking the maximum values for these sizes in both

the horizontal and vertical dimensions of all the individual layout requests. Sizing functions are combined in by simply passing the new size to all of the sizing functions unaltered:

```
> pileWP :: WPlacer
> pileWP lrs = foldr1 pile2 lrs
>    where
>       pile2 (LR mina nata sfa) (LR minb natb sfb)
>          = let newsf rect = do sfa rect
>                                sfb rect
>            in (LR (pairmax mina minb) (pairmax nata natb) newsf)
>       pairmax (x, y) (x', y') = (max x x', max y y')
```

B.4.2 Layout Widgets

This module contains layout combinators and other layout functions.

```
> module LayoutWidgets
> where

> import WidgetCore
```

A widget placer can be used to layout widgets with the `placerW` function, taking a widget placer and a widget as arguments. The widget placer is used to combine the list of layout requests associated with the widget into a single layout request. Finally, a new widget is constructed with a singleton list of layout requests with the combined layout request as its element:

```
> placerW :: WPlacer -> Widget a -> Widget a
> placerW placer (Widget wid)
>    = Widget (\win -> do (a, lrs) <- wid win
>                         return (a, [placer lrs]))
```

A simple widget placer, is the horizontal widget placer. This combines widgets by placing them next to each other horizontally. The minimum and natural sizes of each of the widgets involved are combined by summing the horizontal dimensions and taking the maximum of the vertical dimensions. A new sizing function splits up the allocated screen space between the widgets according to their individual requests. A widget whose natural and minimum sizes are the same will always receive the screen space it asked for, possibly at the expense of widgets whose natural and minimum sizes differ:

```
> horizontalWP :: WPlacer
> horizontalWP lrs = let newmin = foldr1 f (map minSize lrs)
>                        newnat = foldr1 f (map natSize lrs)
>                    in LR newmin newnat (newsf lrs)
>   where
>    f (x, y) (x', y') = (x + x', max y y')
>    diff (LR (x, _) (x', _) _) = x' - x
>    g [] _ _ _ _ _ = return ()
>    g (lr:lrs) offset ay h a b = do let (LR (x, y) (x', _) sf) = lr
>                                        width = if b /= 0 then
>                                                 x' + (((x' - x) * a) `div` b)
>                                                else 0
>                                        width' = max x width
>                                    sf ((offset, ay), (width', max y h))
>                                    g lrs (offset + width') ay h a b
>    newsf lrs ((ax, ay), (w, h)) = do let a = sum (map diff lrs)
>                                          x = sum (map (fst . natSize) lrs)
>                                      g lrs ax ay h (w - x) a
```

The horizontalWP widget placer is used to define the hbox layout combinator:

```
> hbox :: Widget a -> Widget a
> hbox = placerW horizontalWP
```

A widget placer similar to the horizontalWP widget placer, combines widgets by placing them next to each other vertically. The logic behind this widget placer is identical to that for the horizontalWP widget placer, except that the horizontal and vertical dimensions are swapped around. Abstracting away from this leads to the notion of a transpose combinator that would take a widget placer and return a new widget placer identical to the original except that the horizontal and vertical dimensions are flipped. Unfortunately, the choice of representation of layout requests makes it difficult to define such a combinator, as the sizing function is not abstract enough. The sizing function is really a combination of two functions, one splitting a rectangle into a list of rectangles, and the other using this list of rectangles to size the widgets. A transpose combinator needs to be able to modify the list of rectangles before the widgets are sized using them. The modification would flip the horizontal and vertical dimensions. The representation we have chosen, however, does not make the distinction between splitting a rectangle into a list of rectangles, and actually sizing the widgets using this list. As a result of this, we cannot modify the list of rectangles and so cannot write a transpose combinator. Instead we duplicate the logic from the horizontalWP widget placer and manually modify the code to flip the horizontal and vertical dimensions:

```
> verticalWP :: WPlacer
> verticalWP lrs = let newmin = foldr1 f (map minSize lrs)
>                      newnat = foldr1 f (map natSize lrs)
>                  in LR newmin newnat (newsf lrs)
>   where
>    f (x, y) (x', y') = (max x x', y + y')
>    diff (LR (_, y) (_, y') _) = y' - y
>    g [] _ _ _ _ _ = return ()
>    g (lr:lrs) ax offset w a b = do let (LR (x, y) (_, y') sf) = lr
>                                        height = if b /= 0 then
>                                                  y' + (((y' - y) * a) `div` b)
>                                                 else 0
>                                        height' = max y height
>                                    sf ((ax, offset), (max x w, height'))
>                                    g lrs ax (offset + height') w a b
>    newsf lrs ((ax, ay), (w, h)) = do let a = sum (map diff lrs)
>                                          y = sum (map (snd . natSize) lrs)
>                                      g lrs ax ay w (h - y) a

> vbox :: Widget a -> Widget a
> vbox = placerW verticalWP
```

Adding a margin to a widget is a useful layout abstraction, and we provide three functions for doing this. The first adds a margin to the top and bottom of a widget, the second adds a margin to the left and right, and the last places a margin all the way around a widget:

```
> vmargin :: Int -> Widget a -> Widget a
> vmargin m w = vbox (do glue
>                        val <- w
>                        glue
>                        return val)
>               where glue = space (m, m)

> hmargin :: Int -> Widget a -> Widget a
```

```
> hmargin m w = hbox (do glue
>                        val <- w
>                        glue
>                        return val)
>               where glue = space (m, m)

> margin :: Int -> Widget a -> Widget a
> margin m w = hmargin m (vmargin m w)
```

It is sometimes useful to place widgets in a grid, the `matrix` layout combinator can be used for this. The first argument to this layout combinator specifies the width of the grid in terms of the number of widgets placed horizontally on one row. The second argument is a list of the widgets to be used to fill the grid. Combining widgets using this layout combinator results in a new widget, whose return value is a function. This function can be used to obtain the return values of the constituent widgets by passing it a vector indicating which widget in the grid to get the return value of:

```
> matrix :: Int -> [Widget a] -> Widget (Vector -> a)
> matrix n ws
>     = do aws <- vbox (accumulate (map (hbox . accumulate) (splitsegs n ws)))
>          let retwid (x, y) = (aws!!y)!!x
>          return retwid

> splitsegs :: Int -> [a] -> [[a]]
> splitsegs n = takeWhile (not . null) . map (take n) . iterate (drop n)
```

Space widgets have no behaviour, but do take up screen space. Three functions are defined for constructing space widgets. The first constructs a space widget taking up a specified amount of horizontal space, while the second constructs a widget taking up a specified of vertical space. Finally, a combination of these two functions is used to construct a space widget taking up a specific amount of horizontal and vertical space:

```
> hspace :: Int -> Widget ()
> hspace n = Widget (\win -> return ((), [LR (n,0) (n,0) nullSF]))

> vspace :: Int -> Widget ()
> vspace n = Widget (\win -> return ((), [LR (0,n) (0,n) nullSF]))

> space :: Size -> Widget ()
> space (x, y) = do hspace x
>                   vspace y
```

B.4.3 Standard Widgets

This module contains definitions of standard library widgets, such as push buttons, editable text fields, and labels.

```
> module WidgetLib
> where

> import EmbracingWindows
> import WidgetCore
> import LayoutWidgets
```

A button widget is specified by the text label, size of the widget, and a value of type `GUI ()` indicating the behaviour of the button when pressed. The layout request for the widget

indicates that the button can shrink or grow to fit the space it is allocated, since its minimum size is a zero size rectangle. An editable text field widget, and text label widget are constructed in a similar fashion to the button widget. However note, that the layout request for text labels specifies the minimum and natural sizes to be the same indicating that the widget cannot shrink or grow in size:

```
> buttonW :: String -> Size -> GUI () -> Widget PushButton
> buttonW text (w, h) eh
>     = Widget (\win ->
>                   do btn <- mkPushButton win text ((0, 0), (w, h))
>                      onChange btn eh
>                      let sizefun size = setRect btn size
>                      return (btn, [LR (0,0) (w, h) sizefun]))

> editW :: String -> Size -> Widget EditField
> editW text (w, h)
>     = Widget (\win ->
>                   do edit <- mkEditField win text ((0, 0), (w, h))
>                      let sizefun size = setRect edit size
>                      return (edit, [LR (0,0) (w, h) sizefun]))

> textW :: String -> Size -> Widget TextLabel
> textW text (w, h)
>     = Widget (\win ->
>                   do static <- mkTextLabel win text ((0, 0), (w, h))
>                      let sizefun size = setRect static size
>                      return (static, [LR (w,h) (w, h) sizefun]))
```

The concept of an abstract widget that has no visual appearance is quite useful, and a good example of such a widget is a timer widget. A timer widget has no visual appearance, but can be used to perform actions on a regular basis. The widget construction function, `createTimer`, takes a first argument specifying a time interval. The `timerTick` function is used to set the behaviour to occur after this time interval has elapsed:

```
> data TimerControl = TimerControl Window

> timerW :: Int -> Widget TimerControl
> timerW interval = Widget (\win -> do timewin <- mkWindow ""
>                                      setTimer timewin 0 interval
>                                      return (TimerControl timewin, nullLR))

> createTimer :: Int -> GUI TimerControl
> createTimer interval
>     = do timewin <- mkWindow ""
>          setTimer timewin 0 interval
>          return (TimerControl timewin)

> timerTick :: TimerControl -> GUI () -> GUI ()
> timerTick (TimerControl window) handler = onTimer window (\w id -> handler)
```

A stateful text label combines the standard text label widget with a piece of state. The value of type `a` supplied as the third argument specifies the initial value of the state. The widget returns a pair, the first value of which can be used to manipulate the text label, while the second value is a function that can be used to modify the state of the widget:

```
> stateTextW :: Show a => Size -> a -> Widget (TextLabel, ((a -> a) -> GUI ()))
> stateTextW size init
```

```
>   = do st <- stateW init
>        label <- textW (show init) size
>        let update f = do v <- getRef st
>                          let v' = f v
>                          setRef st v'
>                          setText label (show v')
>        return (label, update)
```

A labelled edit widget, as described in Section 3.3:

```
> labelledEditW :: String -> Size -> Widget EditField
> labelledEditW label size = hbox (do textW label size
>                                      editW "" size)
```

B.4.4 Widgets System

This module imports all of the source code that comprises the Widget system.

```
> module Widgets
> where

> import WidgetCore
> import LayoutWidgets
> import WidgetLib
```

B.5 Fudgets

The fudget system implemented here is based upon the original fudgets implementation. The layout combinators and stream processor functions are mostly identical or slightly modified versions of their counterparts from the original fudget implementation. The representation chosen for fudgets is a modified version of that proposed by Reid and Singh, and here the associated combinators are modified versions of the ones they proposed. The implementation of the library widgets is new, although based in some part upon the ideas of Reid and Singh.

B.5.1 Fudgets

This module defines the representation for a fudget and some basic functions for manipulating fudgets.

```
> module FudgetCore
> where

> import EmbracingWindows
```

Layout is handled in the Fudget system in a similar way to the Widget system, using layout requests. A layout request specifies a preferred size and two booleans. The booleans indicate whether the fudget can accept a different size in either of the two dimensions, horizontal and vertical. Again, just as a sizing function was used in the Widget system, a fudget will have an associated sizing function:

```
> type SizeFun = Rect -> GUI ()
> data Layout = L LayoutRequest SizeFun
> data LayoutRequest = Layout Size Bool Bool

> nullSizer :: SizeFun
> nullSizer rect = return ()

> nullLayout :: [Layout]
> nullLayout = [L (Layout (0, 0) True True) nullSizer]
```

Layout combinators in the Fudget system essentially convert a list of layout requests into a single request:

```
> type Placer = [Layout] -> Layout
```

A fudget is represented as a function taking two arguments. The first indicates the window the fudget is to be created inside of, and the second is a handler that can be used to output values from the fudget. The result of the function is a value of the `GUI` monad because the fudget can communicate with the windowing system. The result of the action returned is a pair of values, the first of which specifies an input handler that can be used to send input values to the fudget, the second of which is a list of layout requests:

```
> type Handler a = a -> GUI ()
> type F a b = Window -> Handler b -> GUI (Handler a, [Layout])

> nullHandler :: Handler a
> nullHandler a = return ()

> nullF :: F a b
> nullF parent outputHandler
>     = return (nullHandler, nullLayout)
```

A fudget program can be executed by converting it into a program using the standard Haskell I/O mechanism and then running it. The `fudlogue` function performs this conversion, by passing a null window and null handler to the fudget, and using this as the initialisation for the `startProg` function:

```
> fudlogue :: F a b -> IO ()
> fudlogue f = startProg (do f nullWindow nullHandler
>                            return ())
```

B.5.2 Layout Fudgets

This module contains functions for specifying the layout of fudget programs.

```
> module LayoutFudgets
> where

> import EmbracingWindows
> import FudgetCore
> import Combinators
> import LayoutWidgets      -- For splitsegs
```

The `part` function is useful for dividing a list into two lists determined by a predicate over the elements of the list. This is used in the definition of the `horizontalP` and `verticalP` placers:

```
> part :: (a -> Bool) -> [a] -> ([a], [a])
> part p [] = ([],[])
> part p (x:xs) = let (ys, zs) = part p xs
>                 in if p x then (x : ys, zs) else (ys, x : zs)
```

The following functions are useful auxillary functions for manipulating layout requests:

```
> fixedh :: LayoutRequest -> Bool
> fixedh (Layout _ hf _) = hf

> fixedv :: LayoutRequest -> Bool
> fixedv (Layout _ _ vf) = vf

> minsize :: LayoutRequest -> Size
> minsize (Layout size hf vf) = size

> getRequest :: Layout -> LayoutRequest
> getRequest (L request sizer) = request

> getRequests :: [Layout] -> [LayoutRequest]
> getRequests layouts = map getRequest layouts

> getSizefun :: Layout -> SizeFun
> getSizefun (L _ sizefun) = sizefun
```

The `placerF` function is used to apply a placer to a fudget. It uses the placer to combine the list of layout requests associated with a fudget into a single layout request that is used to build the new fudget:

```
> placerF :: Placer -> F a b -> F a b
> placerF placer f parent outputHandler
>     = do (inputHandler, layouts) <- f parent outputHandler
>          return (inputHandler, [placer layouts])
```

By reversing the list of layout requests before applying a placer, we can reverse the placers action, for example converting a `horizontalP` into a placer that puts fudget next to each other horizontally from right to left instead of left to right:

```
> revP :: Placer -> Placer
> revP placer = placer . reverse
```

Two commonly used placers are `horizontalP` and `verticalP` used to place fudgets next to each other horizontally and vertically, respectively. We define two functions, `hBoxF` and `vBoxF`, that take fudgets as arguments and apply the appropriate placer to make the use of these placers easier. The logic behind the `horizontalP` and `verticalP` placers is similar to that for the Widget system placers. Basically, the available space is split between the fudgets wanting the space according to how much space they each requested. Fudgets that are not willing to accept a change in the space they are given are allocated space first, with the remaining space divided proportionally amongst those fudgets that can accept a change in their size. A transpose combinator is difficult to write for the same reasons as for the Widget system, that is, the sizing functions need to be split into two separate functions. The first of these functions divides allocated screen space amongst fudgets giving a list of rectangles, whilst the second function sizes the fudgets using these rectangles. This would be necessary so that the transpose combinator can modify the list of rectangles produced by the first function. Since the sizing functions are not broken into two in this way, writing a transpose combinator is difficult:

```
> hBoxF = placerF horizontalP
> vBoxF = placerF verticalP

> horizontalP :: Placer
> horizontalP layouts =
>     let requests = getRequests layouts
>         minsizes = map minsize requests
>         h = sum (map getX minsizes)
>         v = (maximum . (0:) . (map getY)) minsizes
>         (fh', fv') = (allf and fixedh, allf and fixedv)
>         allf conn fix = conn (map fix requests)
>         sizer rect =
>             let goth = (fromInt . getX . getSize) rect
>                 gotv = (fromInt . getY . getSize) rect
>                 startx = (fromInt . getX . getPoint) rect
>                 starty = (getY . getPoint) rect
>                 (fih, flh) = part fixedh requests
>                 fixedh' = (fromInt . sum . map (getX . minsize)) fih
>                 floath  = (fromInt . sum . map (getX . minsize)) flh
>                 fixedR = if floath > 0.0 then 1.0 else goth / fixedh'
>                 floatR = if floath == 0.0 then 1.0 else (goth - fixedh') / floath
>                 rR' req = if fixedh req then fixedR else floatR
>                 place x [] = [return ()]
>                 place x (l : ls) =
>                     let req = getRequest l
>                         sf  = getSizefun l
>                         width = (fromInt . getX . minsize) req * rR' req
>                         x' = truncate x
>                         newx = x + width
>                         width' = truncate width
>                     in (sf ((x', starty), (width', gotv)) ) : place newx ls
>             in sequence (place startx layouts)
>     in (L (Layout (h, v) fh' fv') sizer)
```

```
> verticalP :: Placer
> verticalP layouts =
>     let requests = getRequests layouts
>         minsizes = map minsize requests
>         h = sum (map getY minsizes)
>         v = (maximum . (0:) . (map getX)) minsizes
>         (fh', fv') = (allf and fixedv, allf and fixedh)
>         allf conn fix = conn (map fix requests)
>         sizer rect =
>             let goth = (fromInt . getY . getSize) rect
>                 gotv = (fromInt . getX . getSize) rect
>                 startx = (fromInt . getY . getPoint) rect
>                 starty = (getX . getPoint) rect
>                 (fih, flh) = part fixedv requests
>                 fixedh' = (fromInt . sum . map (getY . minsize)) fih
>                 floath  = (fromInt . sum . map (getY . minsize)) flh
>                 fixedR = if floath > 0.0 then 1.0 else goth / fixedh'
>                 floatR = if floath == 0.0 then 1.0 else (goth - fixedh') / floath
>                 rR' req = if fixedv req then fixedR else floatR
>                 place y [] = [return ()]
>                 place y (l : ls) =
>                     let req = getRequest l
>                         sf  = getSizefun l
>                         height = (fromInt . getY . minsize) req * rR' req
>                         y' = truncate y
>                         newy = y + height
>                         height' = truncate height
>                     in (sf ((starty, y'), (gotv, height')) ) : place newy ls
>             in sequence (place startx layouts)
>     in (L (Layout (v, h) fv' fh') sizer)
```

A space fudget has no behaviour but does take up screen space, and is willing to change it's size to fit the available space:

```
> spaceF :: Size -> F a b
> spaceF (w, h) parent outputHandler
>    = return (nullHandler, [(L (Layout (w, h) False False) nullSizer)])
```

Margins can be added to fudgets using the following layout combinators. The first adds a margin to the top and bottom of a fudget, whilst the second adds a margin to the left and right of a fudget. The last adds a margin all the way around a fudget by using a combination of the first two combinators:

```
> vmarginF :: Int -> F a b -> F a b
> vmarginF m f = vBoxF (spaceF (0, m) >*< f >*< spaceF (0, m))

> hmarginF :: Int -> F a b -> F a b
> hmarginF m f = hBoxF (spaceF (m, 0) >*< f >*< spaceF (m, 0))

> marginF :: Int -> F a b -> F a b
> marginF m f = hmarginF m (vmarginF m f)
```

The `noStretchF` function allows the stretchability of fudgets in both the horizontal and vertical dimensions to be changed:

```
> noStretchF :: Bool -> Bool -> F a b -> F a b
> noStretchF fh fv f parent outputHandler
```

```
>    = do (handler, layout) <- f parent outputHandler
>         let layout' = map fixStretch layout
>         return (handler, layout')
>    where fixStretch (L (Layout size _ _) sizer) = L (Layout size fh fv) sizer
```

Combinator layout is implemented in terms of placer layout, using the three placers `horizontalP`, `verticalP` and `revP`:

```
> data Orientation = Above | Below | RightOf | LeftOf

> place2F :: (a -> b -> F c d) -> (a, Orientation) -> b -> F c d
> place2F (><) (f1, a1) f2 = placerF (placer a1) (f1 >< f2) where
>     placer LeftOf = horizontalP
>     placer RightOf = revP horizontalP
>     placer Above = verticalP
>     placer Below = revP verticalP

> (>+#<) :: (F a b, Orientation) -> F c d -> F (Either a c) (Either b d)
> (>+#<)  = place2F (>+<)

> (>==#<) :: (F a b, Orientation) -> F c a -> F c b
> (>==#<) = place2F (>==<)

> (>*#<) :: (F a b, Orientation) -> F a b -> F a b
> (>*#<)  = place2F (>*<)
```

Fudgets can be placed in a grid formation by using the `matrixF` function. This function takes three arguments. The first specifies the width of the grid in terms of fudgets, the second argument is the fudget combinator to use to combine the fudgets, while the last argument is the list of fudgets to use to fill the grid. This layout combinator has a disadvantage as it is currently defined, as the type of its second argument restricts the fudget combinators that can be used to connect the fudgets comprising a matrix:

```
> matrixF :: Int -> (F a b -> F a b -> F a b) -> [F a b] -> F a b
> matrixF n (><) fs = let rows = map (foldr1 colop) (splitsegs n fs)
>                     in foldr1 rowop rows
>                   where colop f1 f2 = place2F (><) (f1, LeftOf) f2
>                         rowop f1 f2 = place2F (><) (f2, Below) f1
```

A fudget is realised inside of a window using the `shellF` function. This function creates a new window and realises the fudget inside of it using the `horizontalP` placer to provide default layout. The layout request of the fudget is used to obtain the desired size for the window. This is used to set the size of the new window. The size for the new window is however, slightly larger than one would expect to account for the size of window borders and menus. The sizing function for the fudget is called to ensure that the fudget is sized appropriately for the new window. An event handler is installed defining the behaviour in response to sizing events to call the sizing function. Finally the newly created window is made visible, and a dummy return input handler and layout request are returned:

```
> shellF :: String -> F a b -> F c d
> shellF title f parent outputHandler
>    = do win <- mkWindow title
>         (_, l) <- (placerF horizontalP f) win nullHandler
>         let [(L (Layout (x', y') _ _) sizer)] = l
>         setWindowRect win ((0, 0), (x' + window_extra_x, y' + window_extra_y))
>         sizer ((0, 0), (x', y'))
```

```
>         onSize win (\w state xy -> sizer ((0, 0), xy))
>         showWindow True win
>         return (nullHandler, nullLayout)
```

B.5.3 Standard Fudgets

This module defines a library of standard fudgets, such as push button fudgets, editable text field fudgets, and text label fudgets.

```
> module FudgetLib
> where

> import EmbracingWindows
> import FudgetCore
```

A useful fudget is one that forces the fudget program to quit when it receives any value on its input stream:

```
> quitF :: F a b
> quitF parent outputHandler
>     = return (inputHandler, nullLayout)
>       where inputHandler a = do quitApp
>                                 return ()
```

A push button fudget can be created by the `buttonF` fudget which takes two arguments, the text label for the button, and the initial size of the button. The code to create the fudget is similar to the `buttonW` function from the Widgets system:

```
> data Click = Click

> buttonF :: String -> Size -> F Click Click
> buttonF text (w, h) parent outputHandler
>     = do btn <- mkPushButton parent text ((0, 0), (w, h))
>          onChange btn (outputHandler Click)
>          let inputHandler a = outputHandler Click
>              sizer size = setRect btn size
>              layout = [(L (Layout (w, h) False False) sizer)]
>          return (inputHandler, layout)
```

A timer fudget outputs the value `Tick` at predefined time intervals:

```
> data Tick = Tick

> timerF :: F (Maybe (Int, Int)) Tick
> timerF parent outputHandler
>     = do return (inputHandler, nullLayout)
>        where inputHandler Nothing = killTimer parent 0
>              inputHandler (Just (interval, delay))
>                  = do setTimer parent 0 interval
>                       onTimer parent (\w id -> outputHandler Tick)
```

An editable text field fudget can be created with the `stringF` function. This uses the data type `InputMsg` to indicate on its output stream any changes that occur to the edit field, such as a simple change, or a commit:

```
> data InputMsg a = InputChange a |
>                   InputDone a
>                      deriving (Eq, Ord, Show)
```

```
> stringF :: Size -> F String (InputMsg String)
> stringF (w,h) parent outputHandler
>     = do edit <- mkEditField parent "" ((0, 0), (w, h))
>          let outputHandler' output = do str <- getText edit
>                                         outputHandler (output str)
>          onChange edit (outputHandler' InputChange)
>          onCommit edit (outputHandler' InputDone)
>          let inputHandler a = setText edit a
>              sizer size = setRect edit size
>              layout = [(L (Layout (w, h) True True) sizer)]
>          return (inputHandler, layout)
```

The `intDispF` function creates a fudget showing the value of an integer. The value being displayed can be changed by sending the new value on the fudget's input stream:

```
> intDispF :: Size -> F Int a
> intDispF (w,h) parent outputHandler
>     = do lab <- mkTextLabel parent "" ((0, 0), (w, h))
>          let inputHandler a = setText lab (show a)
>              sizer size = setRect lab size
>              layout = [(L (Layout (w, h) True True) sizer)]
>          return (inputHandler, layout)
```

The `labelF` function creates a fudget showing a text label. The text label can be changed by sending a new text label on the fudget's input stream:

```
> labelF :: String -> Size -> F String b
> labelF text (w, h) parent outputHandler
>     = do lab <- mkTextLabel parent text ((0, 0), (w, h))
>          let inputHandler a = setText lab a
>              sizer size = setRect lab size
>              layout = [(L (Layout (w, h) True True) sizer)]
>          return (inputHandler, layout)
```

B.5.4 Stream processors

This module defines a representation for stream processors along with the three basic stream processor, `putSP`, `getSP` and `nullSP`.

```
> module SP where

> import FudgetCore

> type SP a b = F a b

> putSP :: a -> SP b a -> SP b a
> putSP init f parent outputHandler
>     = do (inputHandler, _) <- f parent outputHandler
>          outputHandler init
>          return (inputHandler, nullLayout)

> getSP :: (a -> SP a b) -> SP a b
> getSP f parent outputHandler
>     = return (inputHandler, nullLayout)
>        where inputHandler a = do f a parent outputHandler
>                                  return ()
```

```
> nullSP :: SP a b
> nullSP parent outputHandler
>     = return (\a -> return (), nullLayout)
```

B.5.5 Stream Processor Combinators

This module defines some useful combinators for building complex stream processors, along with equivalent infix operators.

```
> module SPCombs
> where

> import Combinators
> import SP
> import SPOps

> infixr 6 >^^=<
> infixl 6 >=^^<
> infixr 6 >^=<
> infixl 6 >=^<

> serCompSP :: SP b c -> SP a b -> SP a c
> serCompSP = (>==<)

> parSP :: SP a b -> SP a b -> SP a b
> parSP = (>*<)

> compSP :: SP a b -> SP c d -> SP (Either a c) (Either b d)
> compSP = (>+<)

> absF :: SP a b -> F a b
> absF x = x

> (>^^=<) :: SP b c -> F a b -> F a c
> sp >^^=< fud = absF sp >==< fud

> (>=^^<) :: F b c -> SP a b -> F a c
> fud >=^^< sp = fud >==< absF sp

> (>^=<) :: (b -> c) -> F a b -> F a c
> f >^=< fud = mapSP f >^^=< fud

> (>=^<) :: F b c -> (a -> b) -> F a c
> fud >=^< f = fud >=^^< mapSP f
```

B.5.6 Stream Processor Operations

This module defines functions for constructing useful stream processors.

```
> module SPOps
> where

> import SP
> import FudgetLib
> import MutVar
```

The identity stream processor simply passes all values that arrive on its input stream to its output stream. A mapping stream processor takes a function that specifies how to generate output stream elements from input stream elements:

```
> idSP :: SP a a
> idSP  = getSP (\x -> putSP x idSP)

> mapSP  :: (a -> b) -> SP a b
> mapSP f = getSP (\x -> putSP (f x) (mapSP f))
```

The `mapAccumlSP` function constructs a stateful stream processor as described in Section 4.4:

```
> mapAccumlSP :: (a -> b -> (a, c)) -> a -> SP b c
> mapAccumlSP f init parent outputHandler
>     = do stateVar <- newRef init
>          return (inputHandler stateVar, nullLayout)
>        where
>            inputHandler stateVar a = do s <- getRef stateVar
>                                         let (s', b) = f s a
>                                         setRef stateVar s'
>                                         outputHandler b
```

The following stream processor filters input stream values to the output stream according to the filter function specified as the first argument:

```
> mapFilterSP :: (a -> Maybe b) -> SP a b
> mapFilterSP f
>     = getSP (\x -> case f x of
>                      Just y -> putSP y (mapFilterSP f)
>                      Nothing -> mapFilterSP f)
```

The `inputDoneSP` stream processor is useful for testing whether an editable text field fudget has had its value commited or not:

```
> inputDoneSP :: SP (InputMsg a) a
> inputDoneSP  = mapFilterSP done
>     where
>         done (InputDone s) = Just s
>         done _ = Nothing
```

The following functions are useful for dealing with editable text field fudgets:

```
> stripInputMsg                :: InputMsg a -> Maybe a
> stripInputMsg (InputDone x)   = Just x
> stripInputMsg (InputChange x) = Just x

> stripInputSP :: SP (InputMsg a) a
> stripInputSP  = mapFilterSP stripInputMsg

> tstInp                 :: (a -> b) -> InputMsg a -> b
> tstInp p (InputChange s) = p s
> tstInp p (InputDone s)   = p s

> mapInp                 :: (a -> b) -> InputMsg a -> InputMsg b
> mapInp f (InputChange s) = InputChange (f s)
> mapInp f (InputDone s)   = InputDone   (f s)
```

B.5.7 Fudget Combinators

This module defines infix operators for combining fudgets.

```
> module Combinators where

> import FudgetCore

> infixr 4 >==<
> infixl 5 >*<
> infixl 5 >+<
```

The `(>==<)` operator combines two fudgets in series:

```
> (>==<) :: F a b -> F c a -> F c b
> (f1 >==< f2) parent outputHandler
>     = do (handler, lra) <- f1 parent outputHandler
>          (inputHandler, lrb) <- f2 parent handler
>          return (inputHandler, lra ++ lrb)
```

The `(>*<)` operator combines two fudgets in parallel, with the values on the input and output streams being untagged:

```
> (>*<) :: F a b -> F a b -> F a b
> (f1 >*< f2) parent outputHandler
>     = do (handler, lra) <- f1 parent outputHandler
>          (inputHandler, lrb) <- f2 parent outputHandler
>          let compInputHandler a = do handler a
>                                      inputHandler a
>                                      return ()
>          return (compInputHandler, lra ++ lrb)
```

The `(>+<)` operator combines two fudgets in parallel, tagging the values on the input and output streams:

```
> (>+<) :: F a b -> F c d -> F (Either a c) (Either b d)
> (f1 >+< f2) parent outputHandler
>    = do (handler, lra) <- f1 parent (outputHandler . Left)
>         (inputHandler, lrb) <- f2 parent (outputHandler . Right)
>         let compInputHandler a = do case a of
>                                        Left v  -> handler v
>                                        Right v -> inputHandler v
>                                     return ()
>         return (compInputHandler, lra ++ lrb)
```

B.5.8 Fudgets System

This module imports all of the source code that forms the fudget system.

```
> module Fudgets
> where

> import FudgetCore
> import LayoutFudgets
> import FudgetLib
> import SP
> import SPCombs
> import SPOps
```

Bibliography

[1] P. Achten and R. Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, January 1995.

[2] R.M. Burstall, D.B. MacQueen, and D.T.Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, University of Edinburgh, 1980.

[3] Magnus Carlsson and Thomas Hallgren. The Fudget Library distribution. Available by anonymous FTP from `pub/haskell/chalmers` on `ftp.cs.chalmers.se.`, 1995.

[4] Jon Fairbairn. Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus. Technical Report 75, University of Cambridge, May 1985.

[5] Sigbjørn Finne. The Haggis distribution. Available by anonymous FTP from `pub/haskell/glasgow/` on `ftp.dcs.gla.ac.uk.`, 1996.

[6] Sigbjørn Finne and Simon L. Peyton Jones. Composing Haggis. In *Eurographics Workshop on Programming Paradigms in Computer Graphics*, April 1995.

[7] Sigbjørn Finne and Simon L. Peyton Jones. Pictures: A Simple Structured Graphics Model. In *Proceedings of the Glasgow Functional Programming Group Workshop*, July 1995.

[8] Sigbjørn Finne and Simon L. Peyton Jones. Concurrent Haskell. In *Proceedings of the Twenty Third ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[9] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer Verlag, May 1995.

[10] Mark P. Jones. The Hugs distribution. Available by anonymous FTP from `nott-fp/languages/hugs` on `ftp.cs.nott.ac.uk.`, 1995.

[11] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages (POPL)*, January 1993.

[12] Donald E. Knuth. *TEX and METAFONT, New Directions in Typesetting.* Digital Press and the American Mathematical Society, 1979.

[13] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, Asilomar, California, 1989.

[14] Rob Noble. *Lazy Functional Components for Graphical User Interfaces.* PhD thesis, Dept. of Computer Science, University of York, November 1995.

[15] Rob Noble and Colin Runciman. Functional Languages and Graphical User Interfaces. Technical Report YCS-94-223.ps.Z, University of York, Department of Computer Science, 1994.

[16] Rob Noble and Colin Runciman. Gadgets: Lazy functional components for graphical user interfaces. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 321–340. Springer Verlag, September 1995.

[17] John Peterson and Kevin Hammond (editors). Report on the Programming Language Haskell, A Non-strict, Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, May 1996.

[18] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[19] Chris Reade and Panos Argiris. A Declarative Toolkit for Graphical User Interface Programming (Using ML with Windows 95). Technical Report CSTR-96-1, Brunel University, 1996.

[20] Alastair Reid. The Yale Release of Hugs. Available by anonymous FTP from `pub/haskell/yale` on `haskell.cs.yale.edu.`, 1996.

[21] Alastair Reid and Satnam Singh. Implementing Fudgets with Standard Widget Sets. In *Glasgow functional programming workshop*, pages 222–235. Springer-Verlag, 1993.

[22] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqness information for graphs. In Schneider and Ehrig, editors, *Proceedings of Graph Transformations in Computers Science, International Workshop*, volume 776 of *Lecture Notes in Computer Science*, pages 358–379. Springer Verlag, 1994.

[23] Mike Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[24] D.A. Turner. Miranda - a non strict functional language with polymorphic types. In *Proceedings IFIP Functional Programming Languages and Computer Architecture*, Nancy France, September 1985.

[25] Tom Vullinghs, Daniel Tuinman, and Wolfram Schulte. Lightweight GUIs for Functional Programming. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 341–356. Springer Verlag, September 1995.

[26] Philip Wadler. Comprehending Monads. In *ACM Conference on Lisp and Functional programming*, pages 61–78, Nice, France, June 1990.

[27] Philip Wadler. The essence of functional programing (invited talk). In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages (POPL)*, January 1992.