Composing Haggis

Sigbjørn Finne* and Simon Peyton Jones

Department of Computing Science, University of Glasgow,
Glasgow G12 8QQ, United Kingdom.
E-mail: {sof,simonpj} @dcs.glasgow.ac.uk

Abstract Haggis is a purely-functional, multi-threaded user interface framework for composing interactive applications. It provides a compositional view of the world of user interface applications, applying to all aspects of the interface the principle of building a component from parts. Interactive components are viewed as *virtual* I/O devices that are composed together to make up complete applications. To fully support this style of programming, Haggis makes good use of the integral features of Haskell, a lazy, purely-functional language. The resulting system offers an extensible, modular and simple programming model for writing user interface applications at a high level of abstraction.

Two key ingredients that Haggis relies on to provide its compositional style are concurrency and monads, making it possible to write multi-threaded functional programs that interact with the Real World comfortably.

1 Introduction

Writing and maintaining user interface software can be a trying experience. Established software engineering habits such as the modularization of code into different parts and the composition of these to make up a complete system, do not currently carry over to the user interface domain [14]. The servitude that the non-user interface parts has to endure under the tyranny of the event-loop is well known, but the frameworks provided for programming the user interface are also lacking. Instantiating and fitting together prefabricated user interface objects is in most cases not too hard, but constructing new *first-class* abstractions *within* these user interface frameworks is not well supported. Rather than building new components by composing existing ones together, the programmer is forced to adopt a completely different and lower level model of programming. The result of having to violate the black box properties of an interactive component is that extending the range of interactive components becomes hard, and programmers stick to the predefined set rather than try to build a component that best fit the interaction at hand.

This paper presents and explores the compositional mechanisms of Haggis (permutation of: A Glasgow Haskell Graphical user Interface System), a functional, multi-threaded user interface framework being developed at the University of Glasgow. Its salient features are:

- *It is based on a functional language.* The framework is implemented in Haskell [9], a lazy, purely-functional language. Working in the context of a high level,

* Supported by Research Scholarship from the Norwegian Research Council

declarative language allow us to make full use of features such as composition, higher-order functions, automatic storage management, static type checking, and the use of monads to structure I/O (Section 2)

- *It is compositional.* The central idea in Haggis is to view the construction of a user interface as a composition from parts. This compositional approach is similar to the object-oriented approaches of Interviews [12], and more recently Fresco [11], but differs in that Haggis provides *a unified programming model for writing both applications and new first-class interactive components*. The compositional mechanisms are described in detail in Section 3, and a discussion of the relationship to other systems can be found in Section 6.

- *It is extensible.* One consequence of using a functional language to provide a compositional style of programming is that writing an user interface application becomes indistinguishable from creating a new user interface abstraction. User defined abstractions are first-class and enjoy the same status as the primitive or predefined components that Haggis happens to already have. Section 4 looks at some of the standard 'devices', while Section 3 deals with the different techniques for extending existing components.

- *It is concurrent.* To be able to treat user interface components as black boxes that can be composed together, Haggis relies on the use of concurrency. It is implemented in Concurrent Haskell [16], an extension of standard Haskell with support for the dynamic creation of lightweight threads and, at the lowest level, shared memory synchronization primitives. Section 3.3 looks at where concurrency is required to maintain a compositional style, and briefly discusses the powerful abstraction technique that the separation of a program into several concurrently running threads offers for user interface applications in particular.

- *Virtual I/O devices.* Interactive objects are the medium by which application and user interact. Haggis extends the metaphor of devices and device handles to such objects, treating interactive components as *virtual* I/O devices where the user and application can exchange information (Section 2.3)

2 Overview

Haskell [9] is the standard non-strict, purely-functional programming language, and several high quality, freely available compilers already exist for it. It differs from mostly-functional languages such as Lisp and SML, in that non-strict languages deliberately do not specify evaluation order, and side-effecting constructs are outlawed.

2.1 Functional Input/Output

Avoiding side-effecting constructs in a functional language is attractive from a semantical point of view, but until recently, at the cost of making the expression of stateful, I/O-intensive programs complex and inefficient. The discovery of the applicability of

monads to functional programming [22] has, amongst other things, provided a framework for writing interactive programs in a non-strict, purely-functional language. For the purposes of this paper, a monad provides a functional framework for expressing computations that side-effect without compromising features such as equational reasoning.

A monad introduces computations or *actions* as values, which can manipulated just like any other value in the language. The fact that a value represents a monadic action is reflected in its type. In the case of the monad used for I/O, such values have type `IO a`. A value of type `IO a` represents an action that, when it is performed, may perform some I/O operations before returning a value of type `a` (lower-case identifiers in type expressions represent polymorphic type variables in Haskell.) So, in the case of simple character I/O operations we have:

```
putc :: Char -> IO ()
getc :: IO Char
```

`putc 'a'` is an action that, when performed, writes its argument to the standard output. Similarly for `getc`, it reads a character from standard input and returns it. Single actions are combined together to make up bigger ones using the following set of basic combinators:

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

`thenIO a1 (\ x -> a2)` joins up two actions in such a way that when performed, the action `a1` is executed first, binding its result to `x` before executing `a2`. `seqIO` is similar, differing only in that the value returned by `a1` is simply thrown away before executing `a2`. `returnIO` is the simplest possible I/O action, as it performs none, just returning the value it was passed!

Armed with these combining forms, I/O 'scripts' can be constructed by stringing actions together. To illustrate, here is the function `getLineIO` which reads a line from standard input (back quoting is the Haskell syntax for infix operators):

```
getLineIO :: IO String
getLineIO =
 getc   `thenIO` \ ch ->
 if (ch==EOF) || (ch=='\n') then
    returnIO []
  else
    getLineIO   `thenIO` \ ls ->
      returnIO (ch:ls)
```

The standard input is read until either the end of file marker or a newline character is encountered, at which point the list of characters on the line is returned. These scripts of actions can be executed using the following two mechanisms:

- Through `main :: IO ()`, which is the function that is first evaluated when a Haskell program is run. It can be thought of as given the state of the Real World, which it then proceeds to side-effect by executing a sequence of actions on it.
- In Concurrent Haskell [16], processes can be created dynamically using the `forkIO` construct (`forkIO :: IO () -> IO ()`.) It eagerly starts to evaluate the action it is passed, concurrently with the context that executed the `forkIO` action.

The key point of the `IO` monad is that the combinators `thenIO` and `seqIO` serialize I/O operations. The evaluation order of the (side-effecting) actions becomes thus fixed, admitting not only an efficient implementation [17], but allows I/O performing, purely-functional programs to be expressed *without* sacrificing vital underlying language properties. Monadic techniques are used when we really want to be explicit the order that we want to perform actions on the outside world.

2.2 Monadic syntax

To aid a monadic style of programming, Haskell 1.3 introduces syntactic support that provides a more familiar style than the Haskell infix notation used above. The syntactic sugar is:

exp	$\rightarrow$	`do` { *stmt* }
stmt	$\rightarrow$	*expr*
	$\mid$	*pat* $\leftarrow$ *expr* `;` *stmt*
	$\mid$	*expr* `;` *stmt*
	$\mid$	*let decls* `;` *stmt*

where exp and pat belong to the syntactic classes of expressions and patterns, respectively. The `` `thenIO` `` and `` `seqIO` ``s are replaced by semicolons, and backarrows are used to bind result values of I/O actions to patterns. As an example, consider this 'sugared' version of `getLineIO`:[1]

```
getLineIO :: IO String
getLineIO =
 do
  ch <- getc
  if (ch==EOF) || (ch=='\n') then
     return []
  else
     ls <- getLineIO
     return (ch:ls)
```

2.3 User interface devices

The monadic I/O model allows the Haskell programmer to express simple file I/O in much the same way as you can in imperative languages, where it is the application

[1] Haskell's layout rules is used here to actually avoid using semicolons and braces to disambiguate the sequence of actions.

that drives the I/O. For example, a program that counts the number of characters in a file would use a loop to read characters from the file one by one until the end of file was reached. A useful advance introduced by UNIX was to present an interface to the program that hid whether the input came from a file, another program or the keyboard.

Haggis extends this device abstraction to include user interface components. This is not new and unique to Haggis [18], but this perspective differs distinctly from an event-driven system. Changing the input of the character counting program in an event-driven system to use a 'virtual keyboard' displayed on the screen would require the program structure to be turned inside out. The interface drives the application. Virtual key presses cause the invocation of action procedures/callbacks to increment the counter; the callback for end of file has to induce whatever actions are meant to follow the counting exercise. Not only is this structure undesirably different from the 'conventional' model, but it is non-compositional; how is a general-purpose counting program supposed to know what to do when a end of file is reached ?

In Haggis user interface components are instead regarded as typed, virtual I/O devices that can be queried, read, written, and closed, just like more conventional ones. Each type of device supports the common set of operations plus a set of device-specific operators, for example a label device supports the setting of a new string label. Representing the interactive components as devices have some advantages:[2]

- *Concrete representation.* In the same way as opening a file returns a file descriptor for the program to subsequently use to access a file, creating user interface components return a *handle* to that virtual device. The handles to virtual devices can be manipulated just like any other value, and new virtual devices can be created by composing existing ones. As a result of the uniform representation, a general-pirpose counting program can now easily be written, just passing it a handle to a device where it can read characters from.
- *Use of application control flow.* The application is in control of the interaction with the virtual devices, meaning that the application flow of control is used to encode the state of the application. This is not the case in an event-driven system, where the application state has to be explicitly updated and maintained between different event handlers.

For each type of user interface device there is a function for creating an instance of it; in the case of a push button:

```
mkButton :: String   -> a -> Widget (Button a)
disable  :: Button a -> Widget ()

pushB :: Widget (Button Bool)
pushB =
 do
  btn <- mkButton "On" True
  disable btn
  return btn
```

[2] We take the word 'device' to encompass both 'normal' devices and interactive components from here on.

`mkButton "On" True` creates a button device labelled `On` that will report the boolean value `True` each time it has been selected. The handle that `mkButton` returns, is used by the application to interact with it. In `pushB`, the button handle is used immediately to disable interaction.

To hide the low level interaction with the underlying window system from the programmer, a monadic abstraction, `Widget`, is introduced. A value of type `Widget a` represents an action that when it is performed, may interact with the underlying window system to create an interactive component, before returning a value of type `a`. Hiding the idiosyncrasies of the window system from the programmer cleans up the code, and avoids accidental 'plumbing' errors.

To make the metaphor of user interface components as virtual I/O devices work in all but the simplest of cases, Haggis and the underlying language has to provide a number of services:

- *Asynchronous forwarding of events to the correct device.* The character counting program does not have to be built around an event-loop, as Haggis takes care of forwarding events such as key presses to the virtual keyboard device. A program operates concurrently, interacting with user interface devices only when it needs to.

- *Support for multi-threaded programming.* The user normally interleaves interaction between different parts of an interface. If the application has to repeatedly check which device was last interacted with and then execute some appropriate action in response, event-loops at the level of devices have effectively been introduced. In Haggis, Concurrent Haskell's [16] lightweight processes are used to dynamically create processes to handle interaction with parts of the user interface. In the character counting example, a separate process can be created to handle interaction with the virtual keyboard, allowing other parts of the application to continue independently.

2.4 Realizing interfaces

To use a device, a program first has to open it. Virtual, interactive devices are realized or opened with `wopen`:

```
wopen :: Widget a -> IO (Window, a)
```

`wopen (mkButton "On" True)` creates a new top-level window containing a button labelled `On`, and returns a handle to the button device. The `Widget` value can be seen as a template, which `wopen` uses to create an instance inside a separate top-level window.

The handle for the window returned by `wopen` is used to reconfigure or close the top-level window, and as an example of how a virtual device can be incorporated into an application, consider the definition of a dialogue box in Figure 1. The message box `alert` consists of two buttons together with a label displaying a message (the operators used to compose the dialogue box are presented at length in Section 3.) To use this alert box abstraction in an application:

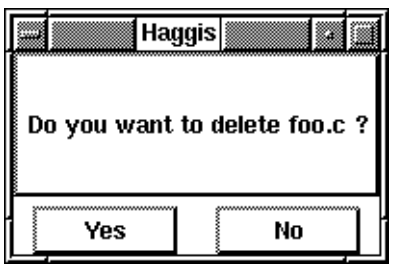
```
mkLabel ::  String -> Widget Label
hBox    ::  Widget a -> Widget a
mkButton::  String -> a -> Widget (Button a)
space   ::  Int -> Widget ()
combineButtons ::  [Button a] -> IO (Button a)
getValue ::  Button a -> IO a

warning str = "Do you want to delete "++str++" ?"

alert ::  String -> Widget (Button Bool)
alert str =
 vBox (
 do
  space 5
  mkLabel (warning str)
  space 10
  hBox (
   do
    space 10
    yes <- mkButton "Yes" True
    space 20
    no <- mkButton "No" False
    space 10
    answer <- ioW (combineButtons [yes,no])
    return answer))
```

Fig. 1: Alert dialogue box

```
safe_delete :: String -> IO ()
safe_delete fname =
 do
  (w,d) <- wopen (alert fname)
  cfirm <- getValue d
  closeWindow w
  if cfirm then
     deleteFile fname
  else
     return ()
```



Executing `wopen` causes the dialogue box on the right to appear on the screen. The application then tries to read the user's response to the delete request, demanding a confirmation before possibly going ahead with the operation.[3] This trivial example

[3] The example code shown here does not create a modal interaction. Other windows on the screen can still be interacted with.

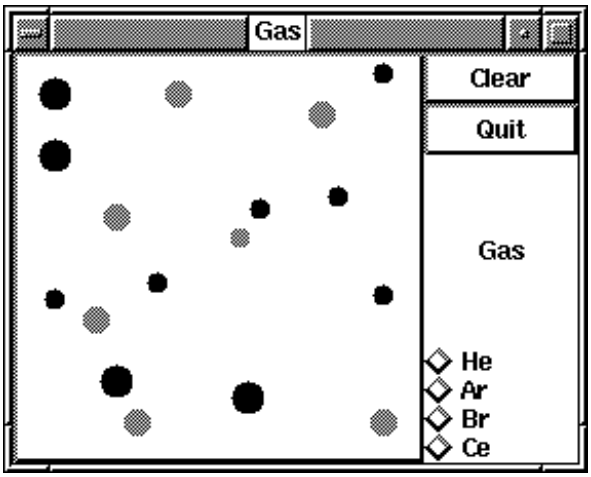


Fig. 2: Gas simulation chamber

highlights some features of Haggis:

- The dialogue box is built by composing together its constituent parts.
- Hiding window system details inside values of type `Widget` avoids esoterica at the level of use, hence there is no need to plumb window system details through the `safe_delete` function.
- Since the delivery of events is performed behind the scenes by Haggis, the program does not hand over control to a centralized run-time system like the event-loop after creating the dialogue box. The `safe_delete` action is free to continue to execute its thread of actions, interacting with the dialogue box only when it has to.

3 Composing the interface

Consider the user interface application in Figure 2, a toy simulation laboratory for visualizing the interaction between atoms in a chamber. Selecting an item in the radio group causes an atom of that type to be inserted into the chamber at a random location with an arbitrary velocity vector. The user can interact with the atoms by grabbing them and throwing them in a different direction, and the chamber can be cleared by pressing a command button. This toy interface is constructed using three separate types of composition:

- At the presentation level, the laboratory is composed out of two parts arranged horizontally, the chamber and the control area. The control area is again constructed out of two separate units, each of which have further internal structure.

Physical composition and how Haggis supports the composition of the visual aspects of a user interface is discussed in Section 3.1.

- The command button for clearing the chamber consist of a graphical output view and a *controller* that attaches interactive behaviour to that view. The controller catches mouse clicks and inverts the button view output and signals the completion of the button press by emitting a value via its device handle. *Behavioural composition* deals with interactive behaviour and how it can be attached to virtual I/O devices to augment or modify their existing behaviour (Section 3.2.)

- At an even deeper level than attributing interactive behaviour to graphical elements, the one-from-many choice provided by the radio group is also a composition. A set of buttons laid out in an arbitrary manner are combined to make up a 'bigger' component that allows only one of them to be selected at a time. *Semantic composition* is concerned with how handles to virtual devices can be combined together, and how application semantic properties such as the exclusive choice of the radio group can be attached to a device (Section 3.3.)

These three types of composition constitute the mechanisms that Haggis offers for building complete user interface applications. We consider each in turn.

3.1 Physical composition

How might we go about describing the visual layout of a collection of components? One way of describing it would be to have operators like:

```
beside  :: [Widget a] -> Widget [a]
```

`beside` takes a list of components, and aligns them all horizontally. The value returned by each component are collected into a list and returned. Unfortunately, this forces all the components laid out with `beside` to have uniform type. Haskell is a statically typed language, so even if `beside` does not inspect the values returned by its components, elements of the list have to be uniformly typed. The result is too tight a binding between interface and application, the physical layout demanding that only components of same type can be arranged. Instead of forcing the programmer to manually coerce the values of the widgets into a common type, we make `Widget a` a monadic type. Operations similar to those provided for gluing I/O actions are provided, this time working on `Widget` actions:

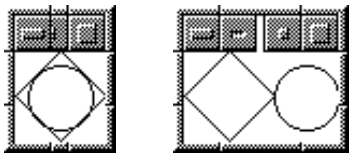
```
thenW   :: Widget a -> (a -> Widget b) -> Widget b
seqW    :: Widget a -> Widget b -> Widget b
returnW :: a -> Widget a
```

`thenW w1 (\ x -> w2)` combines two widget values together to create a larger component, such that when it is realized, `w1` is created first, binding its returning value to `x` before creating `w2`. The default appearance of this composite component is `w1` stacked on top of `w2` in a pile:[4]

[4] Adopting the syntactic sugar of Section 2.2 to describe actions of type `Widget` from here on.

```
pile :: Widget ()
pile =
 do
  mkDiamond 50
  mkCircle 25

row :: Widget ()
row = hBox pile
```



When `pile` is realized, a window with a diamond on top of a circle is created, as shown in the left figure. The combinators `thenW` and `seqW` allow arbitrary components to be piled on top of each other, including, of course, other piles. To get different visual layouts, a pile of components can be constrained by encapsulating it in a *container*. A container spreads the pile out flat, rearranging the components according to the layout scheme it is implementing. The `row` widget applies the `hBox` container to the pile, pulling the circle out from underneath the rectangle in the original pile and aligns them horizontally, as seen on the right. Upon resize of the box, the `hBox` container will calculate the new sizes for its children and resize them accordingly, so whatever internal layout the container has, is hidden from the outside.

What about the coupling between interface and application? The drawback with an operator like `beside` is avoided because the combining together of widgets is separated from attaching a particular layout scheme. Components of different type can be put together in a pile using `thenW` and `seqW`, and if required, later be encapsulated with a container like `hBox`.

Arbitrary layout schemes can be enforced via the encapsulation of `Widgets` inside containers, but by far the most common class of such layout schemes are the tiling operators. Haggis provides a set of tiling combinators based on the TEX model of boxes and glue:

```
hBox,  vBox  :: Widget a -> Widget a
pHBox, pVBox :: Length   -> Widget a -> Widget a
```

A pile of widgets is aligned vertically by `vBox` to construct a *box* which externally appears as one component. Resizing the `VBox` causes the components to be resized, distributing the change in size between them. The combinators `pHBox` and `pVBox` provide the equivalent of the `\parbox` construct in TEX, constraining the length of a box along its axis.

Inter-component void is captured through the component `space` which has no functionality except from laying claim to some screen real estate. The relative willingness of components to both stretch and shrink is finely adjustable through the setting of attributes similar to TEX'. A more substantial example of physical composition is the alert dialogue shown in Figure 1, having the layout structure shown in Figure 3.

For the buttons, an `HBox` is used to place them on a horizontal line, taking care to space them properly. This box is treated as a single component by `VBox`, using only the overall geometry of the line to compute the layout and size of the dialogue box. The structure of the code in Figure 1 reflects quite closely the layout of the realized dialogue

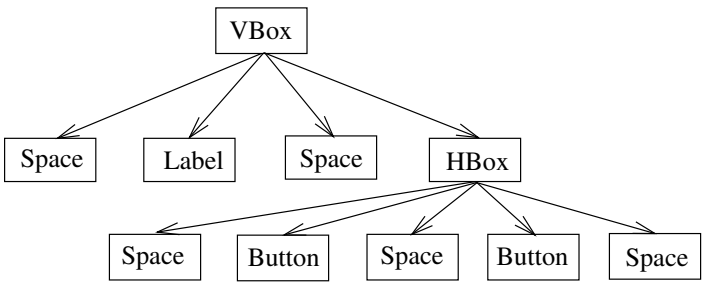


Fig. 3: Layout hierarchy of dialogue box

box, and a good estimate of how a component will be laid out can often be derived simply by looking at the code.

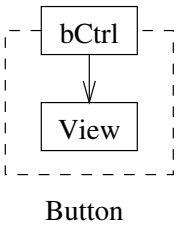
The construction of the physical presentation of a complete interface is achieved in Haggis by repeatedly encapsulating piles of widgets inside containers. A composite component is first-class as seen in the dialogue box example, where an `HBox` was used on equal terms with a primitive label inside a `VBox`.

3.2 Behavioural composition

To make the interface come alive, we need to be able to attach interactive behaviour to the components that were composed physically in the previous Section. *Behavioural composition* is concerned with building new components by adding to or modifying the interactive behaviour of an existing component. The push button of the example in Section 3 was constructed by encapsulating a basic output view inside a controller that catches and translates a mouse click into commands to highlight the output view. A user action such as a key press is ignored by the button controller and just passed on to the view. The basic construct for adding interactive behaviour to a component is `encapsulate`:

```
encapsulate :: (a -> Controller b)
            -> Widget a
            -> Widget b
mkGlyph :: Picture -> Widget Glyph

button :: String -> a -> Widget (Button a)
button str val =
 encapsulate (bCtrl)
             (mkGlyph (frame str))
 where
 bCtrl :: Glyph -> Controller (Button a)
```



`encapsulate (ctrl) comp` constructs a new component where the controlling function `ctrl` is put on top of component `comp`, intercepting all external actions performed on it. The command button is the encapsulation of a view inside the controller

`bCtrl`, as shown on the right. The controller is passed the handle to the view, so that upon seeing mouse clicks it uses the handle to ask the view to change its current output. Note that the type of `encapsulate` is general in that any value of type `Widget a` can be encapsulated, not just simple views. As an example of this, a controller that interprets mouse clicks could also be attached to a general text editing device, so as to allow the mouse to be used for operations such as cursor movement and cutting and pasting.

The controller is also responsible for providing and maintaining an application view of the constructed device. Normally, the encapsulation of a device creates not just a device with modified or augmented interactive behaviour, but also returns a new type of device handle back. `bCtrl` returns a handle to a button device handle that provides the application interface to the new component, i.e. the application can change the button label or listen for button clicks via this handle.

The basic encapsulation or delegation mechanism does not specify how to express the controller itself. Different approaches to specifying interactive behaviour can be accommodated, such as the generic interactors used by [8, 13], where default behaviour can be overridden and adapted to fit context of use, or higher level notations such as the UAN [6]. Haggis does not dictate the manner in which the controllers should be expressed, but we are currently experimenting with an approach similar to that of Interactors[13], where the different types of interactions possible using mouse and keyboard are enumerated. As an example, a simple push button is an instance of a *trigger*, an abstract interaction object that will emit a value when some condition is met.

3.3 Semantic composition

Using the techniques of the previous two sections, hierarchies describing the behaviour and physical layout of interactive devices can easily be constructed. However, the story does not by any means end there. The motivating example in Section 3 used a radio group to provide the selection of an atom type. To express this one-from-many choice externally as a device, the hierarchical, top-down techniques of previous sections do not suffice. Using a controller to encapsulate the items in a one-from-many group wouldn't be a very good solution, for a couple of reasons. Firstly, the controller would have to fix the layout of the items of the group in order to be able to map events such as mouse clicks to individual items. Secondly, it would have to impose the same selection mechanism across all items, making it hard to combine items with different interaction behaviours together in a group. While these restrictions would not be such a high price to pay for a radio group, this top-down solution forces the controller to deal with issues of layout, behaviour and semantics all at once.

Picking an item from many is an operation at the semantic level of devices. The radio group will in response to output reported on any of the item handles, update its current selection and turn off the highlighting of the previous selection. In the case of button devices, this composition or merging of devices to create a composite one is performed by `combineButtons`:

```
combineButtons :: [Button a] -> IO (Button a)
```

```
radio:: [Button Int]
     -> IO (Button Int)
radio ds =
 do
 cdev <- combineButtons ds
 forkIO (rCtrl Nothing ds cdev)
 return cdev

rCtrl :: Maybe Int
      -> [Button Int]
      -> Button Int
      -> IO ()
rCtrl state ds cdev =
 do
  val <- getValue cdev
  case state of
   Nothing ->
    rCtrl (Just v) ds cdev
   Just v ->
    if val==v then
      rCtrl (Just v) ds cdev
    else
      do
       deactivate (devs!v)
       rCtrl (Just val) ds cdev
```

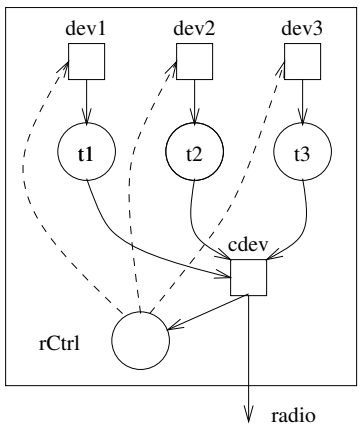


Fig. 4: Exclusive choice group

`combineButtons` takes a list of button device handles and returns a single one. This new device operates by echoing the values that the individual devices output, and is implemented by attaching a separate thread to each device. Each thread is a simple loop that runs independently, reading values from a button device using `getValue` and forwards them to the combined one. The combined device does not implement a radio group though, as the output from a device does not cause the previously selected item in the group to become de-selected. The exclusive semantics of the group can readily be attached though, the code implementing it can be seen in Figure 4.

`radio` takes a list of button devices which have been laid out in some manner, and for simplicity let us assume the numbers which the items output correspond to their position in the list (The restriction in type and value is made here to simplify the code somewhat, more expressive types would normally be used.) `radio` creates a combined device and a process to keep track of the current selection in the radio group, deactivating the previous selection whenever it sees a new one. This is illustrated on the right hand side, where devices are represented as squares and the threads that operate on them are pictured as circles. This example of an exclusive choice shows up some interesting points about Haggis:

- The representation of user interface components as virtual I/O devices was used to compose, using Haskell, a new device by combining existing ones (The implementation for `combineButtons` is not shown here, but it is of the same length and complexity to that of the `exclusive` definition above.)
- The construction of the radio group is not coupled to the user interface at all, as the `radio` function is just an I/O action. The one-from-many choice is now just the merging of input from a set of sources, taking care of notifying input devices that they have become unselected.
- Attaching the `exclusive` function to the combined device is an example of how application semantics can be linked into the user interface. By having a separate process executing `exclusive`, the invariant of only allowing one item to be selected is actively and independently maintained. Constructing an application in Haggis can be seen as the repeated use of such *semantic composition* to link larger and larger pieces of the application into the user interface. Using concurrency, the large and perhaps complex structure of these components can be hidden, as the computation required in response to user interaction will be handled independently to the rest of the application. The result is a modular and extensible application that is not centred around a single event loop.

4 Standard devices

Using the compositional mechanisms just presented, a number of common user interface abstractions have been constructed. To give a flavour of how they appear to the programmer, here are some of the type signatures:

```
mkSlider     :: Widget (Slider Float)
mkHScrollbar :: Float  -> Widget (Scrollbar)
mkField      :: String -> Widget Field
mkViewport   :: Size -> Widget a -> Widget (a, Viewport)
catchMouseEv :: Widget a -> Widget (a, Mouse)
scaleW       :: (Float,Float) -> Widget a -> Widget a
mkPopup      :: Widget a -> Widget (a, Popup)
printerW     :: Widget a -> Widget (a, Printer)
```

The combinator `printerW` encapsulates an arbitrary component for printing, so that when the action `printW printer "dump.ps"`

```
printW :: Printer -> String -> IO ()
```

is executed, a PostScript[5] [1] representation of the current graphical output for the encapsulated component is generated. This is made possible by the device-independent 2D graphics model used to describe graphical output [4], where pictures are described by composing parts, similar to that of [7, 20].

[5] PostScript is trademark of Adobe Systems Incorporated.

5 Implementation

Haggis is operational and currently only available for internal use at Glasgow. It will eventually be released as part of the Glasgow Haskell Compiler(`ghc`), which is available on a wide range of UNIX platforms.[6] Haggis runs under the X Window system (interfaces with Xlib), and uses the concurrency features supported by `ghc`.

Internally, concurrency is used to structure tasks such as the delivery of events, redisplay and the provision of servers to manage resources such as fonts and colours to provide a more convenient and declarative interface to the underlying window system. The windowing model is similar to Fresco's [11], and heavy use is made of lightweight display objects based on glyphs [2], which, in X terms, do not have a window associated with them. A consequence of having such a 'windowless' windowing model is that updates cannot be done by issuing drawing requests to X asynchronously, since a widget could be obscured by others. Rather than having a global redisplay thread which takes care of damage repair, Haggis distributes clipping regions to each component, so that asynchronous redisplay becomes possible.

6 Related Work

Haggis' use of composition as the main programming glue is to some degree used by Fresco [11]. Built on top of class-based, object oriented languages, Fresco provides a set of common user interface abstractions together with a *fixed* collection of operators for combining them. Composition and delegation is used to construct the *graphical* parts of a user interface, so building new abstractions by composing existing ones is possible. However, the model of composition does not extend to the interactive domain, creating new objects requires the abstraction that interactive components represents to be broken and the class hierarchy to be manually extended. Haggis differs in that it tries to provide a uniform programming model where the construction of a user interface is inseparable from the construction of new components. This uniform representation of the world is similar to LiveWorld [21], a prototype-based graphical programming system for experimenting with active objects. It has a uniform object model based on recursive containment, so the hierarchies constructed are similar to what Haggis creates. Although the two systems explore largely different issues, Haggis extends the use of hierarchical composition beyond the interface, providing semantic operations such as `combineButtons`.

The extension of the device abstraction to incorporate interactive components is also used by Acme [18], a concurrent window system that provide access to its windows via a file system interface. The resulting system has a simple, uniform application interface to the windowing capabilities, making its presence almost transparent to the programmer.

Haggis is most closely related to eXene [5], a multi-threaded framework written in Concurrent ML(CML) [19]. The use of concurrency to structure both the underlying implementation and application is common for both systems, but the current implementation of Haggis has a less fine-grained use of processes.

[6] Available by anonymous ftp from `ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/`

Other functional approaches (notably Fudgets [3]) have been evaluated well elsewhere [15]; Haggis differs from these in its use of concurrency and monads to structure interaction with the outside world. Representing user interface components as virtual I/O devices is also a distinguishing feature, and the result is a cleaner separation between interface and application.

7 Conclusions and Future Work

We have in this paper presented Haggis, a compositional approach to user interface construction, describing the various forms of programmer glue that Haggis provides. Interactive components are treated like virtual I/O devices, and three different ways of composing them together were presented:

- *Physical composition.* - the presentational side of an application is described by arranging components in a layout hierarchy.
- *Behavioural composition.* - augmenting the interactive behaviour by encapsulating components inside a controlling layer.
- *Semantic composition.* - building larger semantic units by composing handles of different interactive components. Requires support for concurrency.

One area of further work is to look into ways of defining relationships between different types of interactive devices and operations over them. Haskell's type classes [9] are not expressive enough for defining these relationships, and we are currently investigating how the more powerful type system of constructor classes [10] can be put to use. Alas, it is out of the scope of this paper to go into detail here.

Haggis is operational and one near term goal is to release the system for others to try and evaluate. Although the emphasis is on providing a framework which is extensible through composition of parts, we recognize that a common set of interaction objects has to be provided, and a set of such abstractions is under development.

Currently, Haggis is being used in a compiler environment, and one natural direction of further work would be to integrate Haggis into an interpreter for Haskell, offering a more powerful environment for quickly prototyping and developing user interface applications by combining components together to make up complete applications.

References

[1] Adobe Systems Inc. *PostScript language reference manual*. Addison Wesley, second edition, 1990.

[2] Paul R. Calder and Mark A. Linton. Glyphs: Flyweight objects for user interfaces. In *ACM Symposium on User Interface Software and Technology*, pages 92–101, 1990.

[3] Magnus Carlsson and Thomas Hallgren. FUDGETS – a graphical user interface in a lazy functional language. In *Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture*, pages 321 – 330. ACM Press, 1993.

[4] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995.

[5] Emden W. Gansner and John H. Reppy. eXene. In *Proceedings of the 1991 CMU Workshop on SML*, October 31 1991.

[6] H. R. Hartson, A. Siochi, and D. Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3):181–203, June 1990.

[7] Peter Henderson. Functional geometry. In *ACM Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[8] Tyson R. Henry, Scott E. Hudson, and Gary L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Proceedings of UIST'90*, pages 112–121, 1990.

[9] Paul Hudak et al. Report on the programming language haskell version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[10] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993. ACM Press.

[11] Mark Linton and Chuck Price. Building distributed user interfaces with fresco. In *Proceedings of the Seventh X Technical Conference*, pages 77–87, Boston, MA, January 1993.

[12] Mark A. Linton, J.M. Vlissides, and P.R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[13] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(2):289–320, July 1990.

[14] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, School of Computer Science, Carnegie-Mellon University, July 1993.

[15] Rob Noble and Colin Runciman. Functional languages and graphical user interfaces - a review and a case study. Technical Report 94-223, Department of Computer Science, University of York, February 1994.

[16] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[17] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Conference on the Principles of Programming Languages*, pages 71 – 84. ACM Press, January 1993.

[18] Rob Pike. Acme: A user interface for programmers. In *Proceedings of the Winter 1994 USENIX Conference*, pages 223–234, San Fransisco, 1994.

[19] John H. Reppy. CML: A higher-order concurrent language. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[20] Roger Took. Surface interaction: A paradigm and model for separating application and interface. In *Proceedings of the CHI'90*, pages 35–42, April 1990.

[21] Michael Travers. Recursive interfaces for reactive objects. In *Proceedings of CHI'94*, pages 379–385, Boston, MA, April 24-28 1994.

[22] Philip Wadler. The essence of functional programming. In *Proceedings of the ACM SIGPLAN 19th Annual Symposium on Principles of Programming Languages*, January 1992. Invited talk.