Composing Haggis

Sigbjørn Finne and Simon Peyton Jones

Dept of Computing Science, University of Glasgow
Email: {sof,simonpj}@dcs.glasgow.ac.uk

April 1995

Abstract

Haggis is a purely-functional, multi-threaded user interface framework for composing interactive applications. It provides a highly 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.

Keywords: User interface systems, software composition, functional programming, concurrency, monads.

1 Introduction

Writing and maintaining user interface software can be a trying experience. Established software engineering habits such as the modularisation 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 [16]. 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 pre-fabricated user interface objects together is not too difficult, 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 (a permutation of: A Glasgow Haskell Graphical user Interface System), a functional, multi-threaded user interface framework currently being developed at the University of Glasgow. Its salient features are:

- *It is based on a functional language.* The framework is implemented in Haskell [2], a lazy, purely-functional language. Working in the context of a high level, 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)
- *Virtual I/O devices.* Interactive objects are the medium by which an application and the 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 2.4)

- *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 [14], and more recently Fresco [13], 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 providing a highly compositional style of programming is that the writing of own abstractions becomes easier and more natural. These abstractions are first-class and enjoy the same status as the primitive or predefined components that Haggis happens to 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 [11], an extension of standard Haskell with support for the dynamic creation of lightweight threads and, at the lowest level, shared memory synchronisation 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.

The flexibility and power that composition, in the context of a concurrent, purely-functional language, offers as user interface 'glue' is something that we hopefully will manage to convey during the following sections.

2 Overview

Haskell [2] is the standard language for writing non-strict, purely-functional programs, and several high quality, freely available compilers already exist for it. The language 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 [23] has however, provided a comfortable way of writing I/O-intensive programs in non-strict, purely-functional languages. Most importantly, a monad provides a framework for expressing side-effecting, I/O-intensive programs that does not compromise features such as equational reasoning.

The precise mathematical definition of a monad is not relevant here, the important feature of a monad is that it allows us to express and manipulate computations or *actions* just like any other value in a functional language. The fact that a value represents a monadic action or computation 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`, which reads a character from standard input when performed. 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 very similar, differing in that whatever 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 in a composable manner. 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 `mainIO :: 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 [11], evaluation contexts or threads can be created dynamically using the `forkIO` construct (`forkIO :: IO a -> IO a`.) It eagerly starts to evaluate the action it is passed, concurrently with the context that executed the `forkIO` action.

The inner workings of a monad like `IO` is not relevant here, the key point is that the combinators `thenIO` and `seqIO` serialise I/O operations. The evaluation order of the (side-effecting) actions become thus fixed, admitting not only an efficient implementation [10], but allows I/O performing, purely-functional programs to be expressed *without* sacrificing vital underlying language properties.

2.2 Action syntax

For the purpose of presentation, we introduce some syntactic sugar for expressing monadic actions in a more familiar style than the Haskell infix notation:

$$
\begin{array}{lcl}
\mathit{action} & \rightarrow & \texttt{do}\ \{\ \mathit{action}\ \} \\
 & | & \texttt{return}\ \mathit{expr} \\
 & | & \mathit{pat} \leftarrow \mathit{expr}\ ;\ \mathit{action} \\
 & | & \mathit{expr}\ ;\ \mathit{action}
\end{array}
$$

where *expr* and *pat* belong to the syntactic classes of expressions and patterns, respectively. The following is a 'sugared' version of `getLineIO`:

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

The `` `thenIO` `` and `` `seqIO` ``s are replaced by semicolons, and backarrows are used to bind result values of I/O actions to patterns.

2.3 I/O devices

Actions perform side-effects on the Real World. Normally, we are only interested in operating on a small part of it, so the ability to name the bit that we're interested in becomes important from a programming point of view. So, in the case of operations on files, file descriptors (called *handles* in the I/O proposal for Haskell 1.3) can be created and manipulated. These handles hold the Haskell representation of an external device, and is the medium through which the Haskell program interacts with that device. A file handle is returned as result of opening a file:

```
openFile :: String -> IOMode -> IO Handle
putChar  :: Handle -> Char   -> IO ()
getChar  :: Handle -> IO Char
```

`openFile "foo" ReadMode` tries to open the file `foo` for reading, and if successful, returns the file handle. The file is then subsequently accessed via `putChar` and `getChar`, passing them the file handle representing the file `foo`.

2.4 Virtual I/O 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 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. In an event-driven system, changing the input of the character counting program to use a 'virtual keyboard' displayed on the screen would require the program structure to be turned inside out. The interface now 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 we instead regard user interface components as typed, virtual I/O devices that can be queried, read, written, and closed, just like more conventional ones. Of course, each kind of device will usually support a number of device-specific operations as well. Representing the interactive components as devices have some advantages:[1]

- *Concrete representation.* In the same way as opening a file returns a file descriptor for the program to subsequently use to access the 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.
- *Unifies application interface to external devices.* Handles to external devices now include both 'normal' devices such as files and interactive components. Hence, a general-purpose counting program can now easily be written, just passing it a handle to a device where it can read characters from.
- *Abstract interaction components.* The type of a device determines and constrains the range of operations that the application can perform on it. A push button device would have the abstract type of `Button a`, parameterised on the value it will report when it has been pressed. So an on/off switch would have the type `Button Bool`, returning a boolean to indicate its state. Together with the abstract type of a component, a set of device-specific operations are defined.
- *Device relationships.* To factor out common operations and behaviour, Haskell's *type classes* [2] can be used to define relationships between the different types of virtual I/O devices. For instance, operations to disable or enable the interactive behaviour are meaningful to most interactive devices:

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

```
class Component a where
 disable    :: a -> IO ()
 enable     :: a -> IO ()
 activate   :: a -> IO ()
 deactivate :: a -> IO ()
```

The type class `Component` defines four operations that can be applied to any instance of that type class:

```
instance Component Label where
  disable    = disableLabel
  enable     = enableLabel
  activate   = focusLabel
  deactivate = unfocusLabel
```

The operations defined by `Component` can now be applied to any device that is an instance of its type class. Haskell's type checker will at compile-time infer any illegal uses of these overloaded device types.

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

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

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

`mkButton "On" True` creates a button device labelled `On` that will report the 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 make the button device unselectable.

To create a component like a button clearly requires the underlying window system to be contacted, and to hide the low level nature of this interaction, we introduce the monadic abstraction `Widget`. 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.

In the same way as the operating system takes care of concurrently managing and updating the state of a program's devices, support is required to be able to view interactive components as virtual I/O devices. Haggis takes on the role of an operating system here, providing the following services:

- *Events are asynchronously forwarded to the correct device.* The character counting program does not have to be built around an event-loop, Haggis takes care of forwarding events such as key presses to the virtual device. A program operates concurrently, interacting with a device 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. To be able to separate the character counting task from whatever application it may be part of, Haggis allows you to dynamically create lightweight threads. The thread that executes the

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

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

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

Figure 1: Alert dialogue box

counting loop can then proceed concurrently with any other part of the application the user may be using.

2.5 Realising interfaces

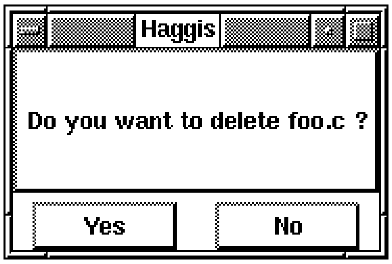
To use a device, a program first has to open it. Virtual, interactive devices are realised or opened with `wopen`:

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

It takes a value of type `Widget` and realises the virtual device on the screen in a separate top-level window, returning a handle to the virtual device and the window it appears inside. A value of type `Widget` is a description of an interactive component, and `wopen` translates it into an interactive object that is put inside a top-level window and mapped onto the screen.

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 device 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 is presented at length in Section 3.) To use this alert box in an application:

```
getValue :: Button a -> IO a

safe_delete :: IO ()
safe_delete fname
 = do {
       (win,dlog) <- wopen (alert fname);
       cfirm      <- getValue dlog;
       closeWindow win;
       if cfirm then
          deleteFile fname;
       else
          return ()
      }
```

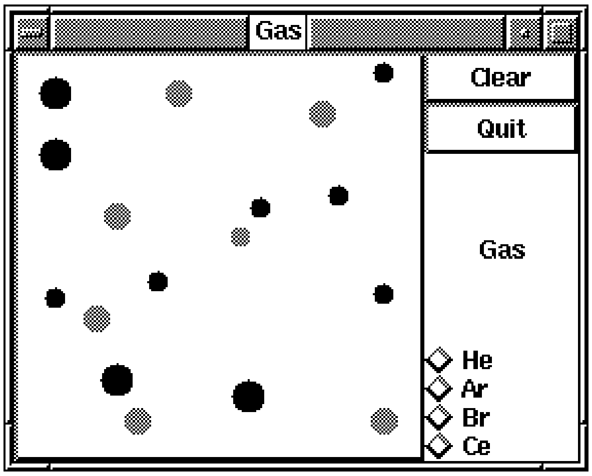


Executing the wopen causes the dialogue box on the right to appear on the screen. The application then tries to read the user's reply to dialogue, requiring the user to confirm the delete operation before possibly going ahead with it. Notable points of this example are:

- 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 centralised 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 following user interface application,



a toy simulation laboratory for visualising 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 with an arbitrary velocity vector and at a random location. 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 built 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 is made out of a graphical output view and a function or *controller* that attaches interactive behaviour to that view. The

controller catches mouse clicks and inverts the button view output plus it registers the completion of the button press in the device handle for the button. *Behavioural composition* deals with interactive behaviour and how it can be attached to virtual I/O devices (Section 3.2), allowing their behaviour to be either augmented or modified.

- 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 such as the items in the radio group can be combined together, and how to link in application code that attach semantic properties such as exclusive choice to these combined devices (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. Describing layout this way has some serious drawbacks:

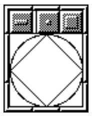
- All the components laid out with `beside` have to have the same type. Haskell is a statically typed language, so even if `beside` does not inspect the values returned by its components, elements of the list have to be uniformly typed. The result is an excessively tight binding between interface and application, the physical layout decreeing that only components of same type can be arranged. The programmer have to perform the manual unification of types through the tedious and error-prone process of tagging types of components.
- Uniformity of types sacrifices the powerful programming tool that types offers in structuring our universe of values and the static type inference of programs to detect any incorrect uses of such values. Writing user interface software is complex enough as it already is, so adding to this by abandoning types altogether is not an attractive proposition.

Fortunately, the excessive binding between interface and application that `beside` induces can be avoided by recalling that values of type `Widget a` are monadic values. Operations similar to those given for glueing I/O actions together, can also be given to `Widget` actions:

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

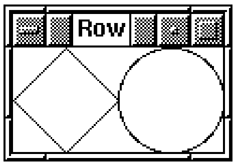
The effect of composing two components together using `thenW w1 (\ x -> w2)` is to create a larger component that when it is realised creates `w1` first, binding its returning value to `x` before creating `w2`. The default appearance of this new constructed component is `w1` stacked on top of `w2` in a *pile*. So when the following component is realised:

```
pile :: Widget ()
pile
 = mkDiamond 20 `seqW`
   mkCircle 10
```



a window with a diamond on top of a circle is created, as shown on the right. 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. For example:[2]

```
row :: Widget ()
row
 = mkHBox
     (do { mkDiamond 20;
           mkCircle 10
         })
```



The `mkHBox` 'pulls' 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.

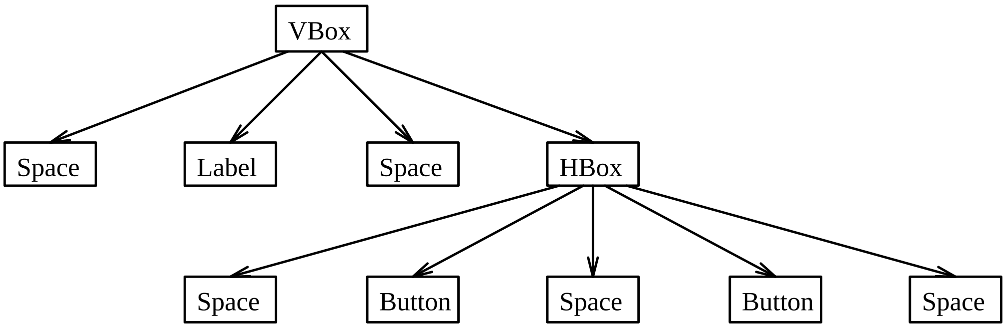
What about the coupling between interface and application? The drawbacks of an operator like `beside` is avoided because the combining together of widgets is separated from attaching a particular layout scheme. Components of different type can be put together in a pile using `thenW` and `seqW`, and if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

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:

```
mkHBox  :: Widget a -> Widget a
mkVBox  :: Widget a -> Widget a
mkPHBox :: Length   -> Widget a -> Widget a
mkPVBox :: Length   -> Widget a -> Widget a
```

A pile of widgets is aligned vertically by `mkVBox` 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 `mkPHBox` and `mkPVBox` provide the equivalent of the `\parbox` construct in TEX, constraining the length of a box along its axis.

Inter-component void or space 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'. To give a more substantial example of physical composition, let us examine the hierarchical layout structure of the alert dialogue box in Figure 1:



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 realised dialogue 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

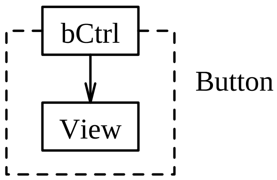
[2] We adopt the syntax of Section 2.2 to describe actions of type `Widget` from here on.

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* constructs new devices by adding to or modifying the interactive behaviour of a device. In the case of the command button of Section 3, it was constructed by encapsulating an 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 combining an interactive device and a controller is `encapsulate`:

```
encapsulate :: (a -> Controller b)
            -> Widget a
            -> Widget b

button :: String -> a -> Widget (Handle a)
button str val
 = encapsulate (buttonCtrl str val)
               (mkView (frame str))

mkView :: Picture -> Widget View

buttonCtrl :: String
           -> a
           -> View
           -> 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 `buttonCtrl`, 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.

A 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 back. Hence, the controller for the button also needs to construct and return a handle to a device that can be used to perform operations on it, operations such as changing the button label or disabling it. When the application performs these requests, the controller is responsible for converting them into actions on the encapsulated device.

The basic mechanism for encapsulating a component inside a controller does not specify the manner in which the controller should be expressed. Several approaches to specifying the interactive behaviour can be accommodated, such as the generic interactors used by [15, 7], where default behaviour can be overridden and adapted to fit context of use, or higher level support in the form of 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[15], where the different types of interactions possible using mouse and keyboard are attempted enumerated.

What is the difference between the hierarchical construction of controllers and the repeated encapsulation of sets of components inside containers to build the physical presentation of the interface? Although the resulting structures are similar, these two composition mechanisms address largely different issues. The containers control and manage the geometries of its children and are not concerned with user actions such as key presses, while behavioural composition addresses the problem of attributing interactive behaviour to existing components, and lower level requests such as resizes are normally not of interest to them. However, there are cases where the distinction between physical and behavioural composition do become blurred:

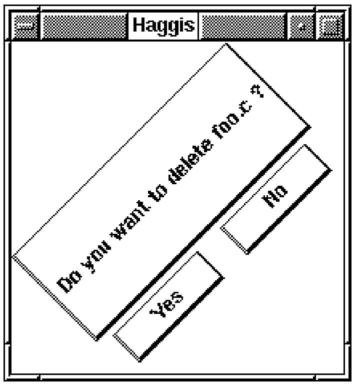
- *Viewports.* A viewport is created by attaching to a simple view a controlling function that will manage an arbitrary complex component inside it by providing a mapping from the coordinate space of the viewport to the coordinate space of the encapsulated component:

```
mkViewport :: Size -> Widget a -> Widget (Viewport,a)
```

  `mkViewport (100,100) app` creates a view of initial size `(100,100)` that displays the interactive device `app` inside it. The viewport manages both the display of a sub-view of `app`, and the transformation of coordinates contained in events such as mouse clicks from the viewport coordinate system into the coordinate system of the viewed object. Note that there is nothing to stop you from having arbitrary levels of viewports, so you can easily have applications embedded inside viewports, which again have applications running inside their own viewports and so on.
- *Transforms.* Haggis also has a compositional 2D structured graphics model [4], where geometric transformations such as scaling and rotation can be applied to any graphic object. Extending the support for such transformations to interactive devices is straightforward, and combinators like the following are provided in Haggis:

```
rotateW :: Degrees -> Widget a -> Widget a
scaleW  :: (Float,Float) -> Widget a -> Widget a
```

  `rotateW 45 (alert "foo.c")` will display the dialogue box at angle of 45 degrees:



  Achieving the effects of [20], but , alas, most window managers don't support the rotation of top-level windows!
- *Pop-up views.* Similar to viewports, pop-up views can be provided that encapsulates a component inside a pop-up window which can be selectively mapped onto the screen:

```
mkPopup :: (Int,Int) -> Widget a -> Widget (Popup, a)
showPopup :: Popup -> IO ()
hidePopup :: Popup -> IO ()
```

The modularisation of user interface objects into output views, controllers and application objects is a common technique of achieving separation of concerns between interface, application and a mediating layer [12, 8]. *Behavioural composition* deals with the mechanism for attaching behaviour to existing components without having to leave the compositional model. It does not address the issue of how applications and their semantics are linked up to these interactive components, that is the subject of the next section.

3.3 Semantic composition

Using the techniques of the previous two sections, hierarchies describing the behaviour and physical layout of interactive devices can be built. 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 from a collection. To express this one-from-many choice externally as a device, the hierarchical, top-down techniques of previous sections do not suffice. To see this, let us attempt to define a controller that encapsulates the items of the radio group. It implements the one-from-many choice by keeping track of the current selection, so that when it sees mouse clicks it un-highlights the current selection and highlights the new one. This solution is not very satisfactory, for the following reasons:

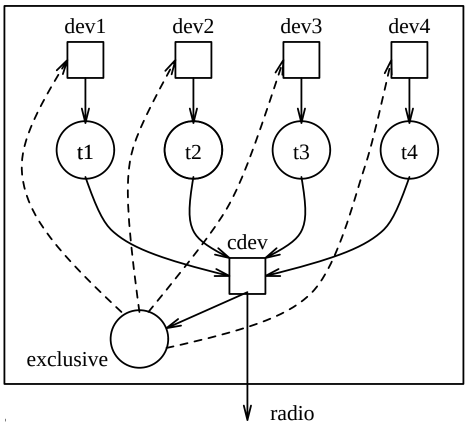
- The physical layout of the items have to be fixed by the controlling function, as it has to be able to map actions such as mouse clicks to individual items.
- The user actions required to select individual items is fixed by the controlling function, making it difficult to combine different interaction behaviours into the same group.

These two restrictions may not be such a high price to pay for the radio group in question, but this monolithic solution is not very satisfactory in general as it binds the choice too close to its presentation. Picking an item from many is really an operation at the semantic level of devices. The radio group will in response to output reported on either of device handles for the items in the group, update its current selection and ensure that the previous selection is un-highlighted. This composition or merging of devices to create a composite one is performed by `combine`:

```
getValue :: Button a -> IO a

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

`combine` 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 which reads 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:

```
radio :: [Button Int] -> IO (Button Int)
radio devs
 = do { cdev <- combine devs;
        forkIO (exclusive Nothing devs cdev);
        return cdev}

deactivate :: Button a -> IO ()

exclusive :: Maybe Int
          -> [Button Int]
          -> Button Int
          -> IO ()
exclusive Nothing devs cdev
 = do { v <- getValue cdev;
        exclusive (Just v) devs cdev}
exclusive (Just v) devs cdev
 = do
    { new_v <- getValue cdev;
      if (new_v == v) then
         exclusive (Just v) devs cdev
      else do
         { deactivate (devs!!v);
           exclusive (Just new_v) devs cdev}}
```



`radio` takes a list of button devices which have been laid out in some manner, and for simplicity let us assume the `Ints` that 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 creates a thread that keeps track of the current selection in the radio group, deactivating the previous selection whenever it sees a new one. This is illustrated to the right, where devices are represented as squares and the threads that operate on them are pictured as circles. This simple example shows up some important features of 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 `combine` is not shown here, but it is very similar to 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 thread that executes `exclusive`, the invariant of only allowing one item to be highlighted is maintained and the compositional structure of the radio group is completely hidden from the outside. 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. My making the application code that is linked in execute concurrently, centralisation of an application around a single dispatch loop is avoided. Device input will now be seen by the attached application 'immediately', as it runs in a separate thread. The resulting program is modular and can easily be modifed or extended.

4 Standard devices

Haggis provides a number of standard devices that implement common user interface abstractions:

Selectors are devices that have an on/off state, and encompasses various types of buttons that respond to mouse clicks and devices like menu items that are activated on mouse pointer movements. Selectors that have different appearance or activation events are straightforward to create by combining together an output view with an appropriate controller.

Graphics The graphical output is described using a structured graphics model similar to that of [21], where abstract pictures are described by composing parts (see [4] for details.) Three basic devices are provided for displaying graphics:

Glyph the most basic output view, manages the display of a single picture.

Canvas is a device that manages a collection of pictures, which can be added to or deleted from dynamically.

Printer a composite device that encapsulates an arbitrary component inside a controller. When the printer device is told to print, it will generate PostScript for the component it is encapsulating.

Sliders Haggis defines both one-dimensional and two dimensional sliders that can be used to select a value from within a user-defined range.

Scrollbars by composing together sliders and buttons, scrollbars are provided.

Text the input and output of text is support via devices for that allow strings and arbitrary text to be viewed and edited.

5 Implementation

Haggis is currently being implemented actively, and at the time of writing, the framework for composing components is in place together with a smallish collection of common user interface components. The framework runs under the X Window System (interfaces with Xlib) and is implemented using the threads extensions of the Glasgow Haskell Compiler(`ghc`), which is available on a wide range of UNIX platforms.[3]

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

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 (more details on Haggis' internal architecture and how it is implemented in Haskell, can be found in [3].) The windowing model is similar to Frescos, and heavy use is made of lightweight display objects based on glyphs [1], 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 requests to X directly, but have to be done via a separate top-level redisplay function that traverses the component hierarchy in the correct order. In Haggis, this task is assigned to a separate thread that walks over the hierarchy at regular intervals, updating the output where necessary. To avoid unnecessary flicker, double buffering techniques are used, rendering into an off-screen pixmap before copying it to the output window. One result of delegating the rendering task to a separate redisplay thread is a more declarative solution where output views operate on abstract descriptions of pictures, leaving the dirty work of converting them into actual X requests to the redisplayer. The calling interface used by the redisplay thread to render a the output of a device is render-independent, and it is used to generate PostScript output when you want to print the user interface.

6 Related Work

Haggis' use of composition as the main programming glue is to some degree used by Fresco [13]. 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 encapsulation of interactive components to be broken and the class hierarchy to be manually extended. Haggis 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 [22], 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 very similar to what Haggis creates. Although the two systems explore largely different issues, Haggis extends the use of hierarchical composition beyond the interface, providing operations such as `combine` to build application semantic constructions.

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 threads. Assigning a thread to all separable tasks in a user interface has not only resource implications, but quite often the relationships between components can be expressed without the use of concurrency. An example is controller-view pairs, where the view's only source of input is from the controller. The construction can be viewed as a server that in response to input from user or application may reply via the view or by returning some value via its handle. The requests are handled sequentially, so the interaction between controller and view is synchronous. Having the two run as separate threads is thus strictly not necessary, their relationships could be expressed using functions and function application. We are currently looking into where concurrency is appropriate in a user interface.

Other 'functional' approaches have been evaluated well elsewhere [17]; Haggis differs from these in that it treats user interface components as virtual I/O devices and through the use of concurrency and monads is able to provide a cleaner separation between interface and application in a user interface application.

7 Conclusions and Future Work

We have presented in this paper 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* - constructing the visual layout by arranging the components in a hierarchy of boxes/containers.
- *Behavioural composition.* - attaching or augmenting the interactive behaviour of components through encapsulation of a controlling function.
- *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 more comprehensive relationships between different types of interactive devices and operations over them. Haskell's type classes goes some way in providing a framework for doing this, but fails for an operator like:

```
getValue :: Button a -> IO a
```

The value returned by `getValue` is dependent on `Button`'s type parameter, a relationship that cannot be expressed using type classes, but requires the more powerful type system extension of constructor classes [9].

Haggis is currently under construction, 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 basic parts, we recognise that a common set of interaction objects has to be provided, and a set of such abstractions is currently under construction.

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

References

[1] Paul R. Calder and Mark A. Linton. Glyphs: Flyweight Objects for User Interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 92–101, 1990.

[2] Paul Hudak et al. Report on the Programming Language Haskell Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[3] Sigbjorn Finne and Simon Peyton Jones. The ingredients of haggis. in preparation, April 1995.

[4] Sigbjorn Finne and Simon Peyton Jones. Pictures - Structured Graphics in Haskell. in preparation, April 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] 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.

[8] Ralph D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *Proceedings of CHI'92*, pages 335–342, Monterey, California, May 1992.

[9] 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.

[10] 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.

[11] Simon Peyton Jones and Sigbjorn Finne. Concurrent Haskell. in preparation, April 1995.

[12] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[13] 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.

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

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

[16] 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.

[17] 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, Univeristy of York, February 1994.

[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] Steven H. Tang and Mark A. Linton. Blending Structured Graphics and Layout. In *Proceedings of ACM Symposium on User Interface Software and Technology*, November 1994.

[21] Roger Took. Surface Interaction: A Paradigm and Model for Separating Application and Interface. In *Proceedings of the CHI'90*, pages 35–42, April 1990.

[22] Michael Travers. Recursive Interfaces for Reactive Objects. In *Proceedings of CHI'94*, pages 379–385, Boston, MA, April 24-28 1994.

[23] 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.