FranTk – A Declarative GUI System for Haskell

Meurig Sage

Chapter 1 - The Basic Concepts

1.1. Introduction

Developing a GUI in Haskell should be easy. Yet despite a whole slew of systems, it's still difficult to knock up a quick interface, let alone develop a complex, dynamic one. A range of different paradigms have been suggested, based on callbacks or full concurrency. They've all got their problems. Building a complex system can result in a mess of callbacks and mutable references that can seriously confuse the structure of your program. Concurrent programming can help with some of this structure, but encourages a whole range of alternative problems associated with mutual exclusion and multiple processes.

The other major problem that we face is producing convincing looking, platform independent code. Without the proper support this too can be a bit of a nightmare.

With FranTk (pronounced "frantic"), life should (hopefully) be a bit simpler. It's designed to allow you to build your application in a declarative manner. It uses behaviours and events, concepts from Conal Elliot's Functional Reactive Animation. These allow you to model a system over time. Events are used to describe values that occur discretely, such as button clicks. Behaviors are continuous quantities that vary over time. They are used to represent the state of an application. Events and behaviors can interact. For instance, we can have a behavior that changes on every event. You can then render this application on to an interface.

FranTk currently lives on top of Tcl-Tk. This provides a platform independent and robust language for building GUIs. FranTk code will run unchanged on Windows and Unix, giving native look and feel on each system. However, it has been designed to be as Tcl-Tk independent as possible. This means it can be ported to other GUI systems. We are currently also investigating a version using Java's swing libraries. But enough marketing, let's get on with a few examples.

1.2. First Example

Let's start with a quick example to get across some basic concepts. The "Hello World" program is defined as follows:

```
main :: IO ()
main = start $ render withRootWin $ mkLabel [text "Hello World"]
```

(Here $ is used for function application, f $ b = f b. This cuts down on the number of brackets necessary)

Let's break that down. We create a label widget, with the text "Hello world". The type signatures are as follows:

```
mkLabel :: [Conf Label] -> Component
text :: Has_text w => String -> Conf w
```

We use `mkLabel` to make a label. It takes a list of configuration information, in this case, some text to display. As with TkGofer we use type classes to guarantee that only the correct configuration information can be applied to any widget. The `text` function takes a String and returns a configuration option that can be applied to any object that is a member of the `Has_text` class. This class includes labels, as they are capable of displaying text.

The `mkLabel` function returns a `Component`.

```
type Component = GUI Widget
```

A Component is an action that produces a `Widget`. This uses the GUI monad, which is an extension of the standard IO monad. Values of type `GUI a` represent actions that may have some side effect on the user interface, such as creating a label, and return a value of type **a**.

A `Widget` is an abstract data type representing primitive Tcl widgets. A Widget may in fact be made up of several primitive tcl widgets, and may be dynamic, changing its appearance over time.

We now have to run the component within a window. In this example we run the component in the root window, using `withRootWindow`.

```
withRootWindow :: [Conf Window] -> Component -> WComponent
```

This takes a list of configuration options for a window, and produces a `WComponent`. This is an action that produces a `WindowWidget`, which is an abstract representation of a window (that will contain widgets).

```
type WComponent = GUI WindowWidget
```

We have now distinguished two different types of widgets

- Standard widgets, such as buttons and labels, that can be composed with operators such as above and beside;
- Window widgets that represent actual windows

We will come across other types of widget later.

Now we need to render this window component onto the screen. We do this using `render`.

```
render :: WComponent -> GUI ()
```

Finally, to run the GUI actions that we have produced we use start. This runs the action and then starts up the tcl-tk event loop. This event loop will run until the graphical user interface quits, at which point it will return.

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

As `start` and `render` are often used together there is a composite function display.

```
display :: WComponent -> IO ()
display = start . render
```

1.3. Interactive Example I – A Simple Counter

Now let's try a simple interactive example that shows how you handle state in FranTk. Consider the simple interface below.

It shows a simple counter with a label, an increment and decrement button, and a slider (known as a scale widget). This widget has a current value shown by the slider and the label. Pressing the increment or decrement button, or moving the slider will change this value. We therefore have multiple views of some data.

1.3.1. Representing State with BVars

To represent the state in the example we use a `BVar`. A value of type `BVar a` represents some abstract mutable state of type a.

```
data BVar a
```

We can create a new `BVar` within the `GUI` or the `IO` monad. Most commonly we will be using them within the `GUI` monad.

```
newBVar :: a -> IO (BVar a)
mkBVar :: a -> GUI (BVar a)
```

We'll come across other forms of `BVar` later that can be created within the `IO` or `GUI` monad. In general we use the naming policy that the IO version is prefixed with `new` and the `GUI` version with `mk`.

It is possible to get a behavior from a Bvar.

```
bvarBehavior :: BVar a -> Behavior a
```

A value of type `Behavior a` is a time varying value of type `a`. The behavior therefore represents the value of the counter at any give point in time.

It is possible to get an event from a `BVar`

```
bvarEvent :: BVar a -> Event a
```

An event is a stream of *occurrences*, each of which has a specific time and value. The type `Event a` denotes an event that generates a value of type `a` when it happens. The event from a `BVar` therefore generates an occurrence every time the value of the `BVar` changes.

In our example we would therefore represent the state of the counter as a value of type `BVar Int`.

1.3.2. Using the value of a BVar

What can we do with a behavior? We can tell the label and slider to display the behavior values that we get from the `BVar`.

```
lbl :: BVar Int -> Component
lbl m = mkLabel [textB (lift1 show (bvarBehavior m))]

textB :: Has_text w => Behavior String -> Conf w

lift1 :: (a -> b) -> Behavior a -> Behavior b
```

We can tell a label to display a `String Behavior` using the `textB` configuration option. We can turn the `Integer Behavior` (of the `BVar`) into a `String Behavior` using `lift1`, which maps a function across across the `Behavior`. Further combinators exist to compose several behaviors, such as `lift2` shown below.

```
lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
```

We can tell the slider to use the value of the `BVar` with `scaleValB`. This sets the value of the slider to that of an integer behavior. (We will fill in the rest of the definition (the .. part) later.)

```
scale :: BVar Int -> Component
scale m = mkHScale [scaleValB (bvarBehavior m)] (..)

scaleValB :: Behavior Int -> Conf w
```

1.3.3. Setting the value of a BVar

We can set the value of a `BVar` using its `Listener`.

```
bvarInput :: BVar a -> Listener a
bvarUpdInput :: BVar a -> Listener (a -> a)
```

A listener is an abstract type, but it can be thought of as `Listener a = a -> IO ().` A value of type `Listener a`, is a function, that given a value of type a, performs a side-effecting `IO` action with it. Listeners are therefore consumers of values.

The listener accessed by `bvarInput` updates the `BVar` to its given value. This will alter the value of the `BVar's` behavior and generate an event occurrence. The listener accessed by `bvarUpdInput` updates the `BVar` by applying the given function to its current value.

We can therefore define the increment button as follows.

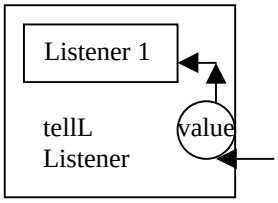
```
incb :: BVar Int -> Component
incb m = mkButton [text "inc"]  (tellL (bvarUpdInput m) (+1))

mkButton :: [Conf Button] -> Listener () -> Component
```

The function `mkButton` takes a list of configuration information for a button. This defines its appearance. Note that Button is also an instance of the `Has_text` class. It also takes a listener which is passed the value () every time the button is pressed.

We therefore need to make the button talk to the listener provided by the `BVar`. We do this using `tellL`.

```
tellL :: Listener a -> a -> Listener b
```

The function `tellL` takes a listener and a value, and returns a listener that ignores its argument and always performs its action with this value. (`tellL` is one of a number of listener functions described in chapter 2).



In the definition of incb we therefore produce a listener that that ignores its argument and always updates the `BVar` using the function (+1).

The slider simply updates the `BVar` with its changed value. The slider's listener is fired with its current value every time the slider is moved. We use `mkHScale` to make a horizontal slider.

```
scale m = mkHScale [..] (bvarInput m)
mkHScale :: [Conf Scale] -> Listener Int -> Component
```

1.3.4. Composing Components

Finally we compose the components

```
counterB :: BVar Int -> Component
counterB m = above (lbl m) (beside (incb m) (decb m))

composite :: BVar Int -> Component
composite m = above (counterB m) (scale m)
```

Note that above and beside are used here to compose components.

```
above :: Component -> Component -> Component
beside :: Component -> Component -> Component
```

The complete code for the example is therefore:

```
main :: IO ()
main = display $ withRootWindow [] $ scaleAndButton

scaleAndButton :: Component
scaleAndButton = do {m <- newBVar 0; composite m}

composite,scale,counterB,lbl,incb,decb :: BVar Int -> Component

composite m = above (counterB m) (scale m)

scale m = mkHScale [scaleValB (bvarBehavior m)] (bvarInput m)
counterB m = above (lbl m) (beside (incb m) (decb m))

lbl m = mkLabel [textB (lift1 show (bvarBehavior m)]
incb m = mkButton [text "inc"] (tellL (bvarUpdInput m) (+1))
decb m = mkButton [text "dec"] (tellL (bvarUpdInput m)
                                      (subtract 1))
```

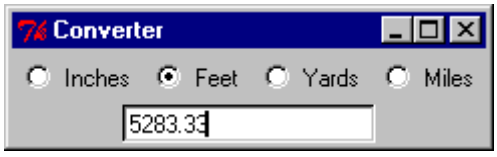
1.4. A brief aside - The Has_Event class

In fact, `BVar`, along with `Wire`, which we will come across later, is a member of a more general class called `Has_Event`. This provides access to its listener and event.

```
class Has_Event c where
 input :: c a -> Listener a
 event :: c a -> Event a

instance Has_Event BVar where
 input = bvarInput
 event = bvarEvent
```

1.5. Interactive Example II – A Distance Converter

To illustrate some other important concepts we'll look at a second interactive example. Consider a distance converter. It can operate in inches, feet, yards and miles. Distances are typed into an entry area; pressing the return key sets the current value. Changing the mode causes the distance to be converted to the alternate units.



We'll now implement this converter. To make a converter we pass in a `BVar` representing conversion mode, which consists of a pair of functions to translate to and from standard units; a `BVar` representing the current value of the converter in standard units; and a list of convertion modes with their names.

```
type Convert = (Double -> Double,Double -> Double)

converter :: BVar Convert -> BVar Double -> [(String,Convert)]
          -> Component
```

1.5.1. Using Text Entries

The converter consists of an entry and a set of radio buttons. We'll deal with the entry first. We create the entry itself using the function `mkEntryRtrn`.

```
mkEntryRtrn :: [Conf Entry] -> Listener String -> Component
```

This expects a list of entry configuration information and a listener. It tells the listener the current value of the entry every time that the return key is pressed.

The entry displays the current value of the converter in terms of the current units. We can create this value with the function `currentValue`.

```
currentValue :: BVar Convert -> BVar Double -> Behavior String
currentValue state value =
          lift2 getVal (bvarBehavior state) (bvarBehavior value)
  where getVal :: Convert -> RealVal -> String
        getVal (toUnit,fromUnit) unit = show $ fromUnit unit

lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
```

Note that we use `lift2` here to compose two behaviors with a mapping function to make a third behavior.

1.5.2. Sampling behaviors

When the return key is pressed the converter needs to take the new value from the text entry, convert from current units into standard units, then set the value `BVar` with this value. We need to sample the current mode to do this. The `mkEntryRtrn` function expects a listener that will be told about string values. The `setValue` listener is therefore implemented as follows.

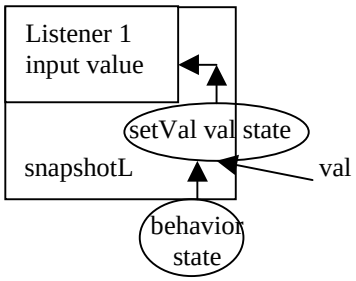
```
setValue :: BVar Convert -> BVar Double -> Listener String
setValue state value =
    withSnapL setVal (bvarBehavior state) (input value)
  where
    setVal :: String -> Convert -> RealVal
    setVal val (toUnit,fromUnit) = toUnit $ read val

withSnapL :: (a -> b -> c) ->Behavior b ->Listener c -> Listener a
```

We use `withSnapL` to make a listener that snapshot a behavior and consumes its current value. Its first argument is a function that composes the value passed to the listener and the value of the behavior to create a new value.

Note the funny looking type. This is because listeners are consumers of values not producers. If we think of `Listener` as being of type `a -> IO ()`, then, given some `IO` function sample, that samples a behavior, we can think of the definition of `withSnapL` as follows:

```
sample :: Behavior a -> IO a

withSnapL f b l = \x -> do {v <- sample b;l (f x b)}
```

The `setValue` listener, formed from `withSnapL`, consumes Strings, samples the *current mode behavior*, applies `setVal` to the String and mode to get a new distance, and tells the *value BVar* about this distance.



Finally, we can define the entry as follows:

```
entry :: BVar Convert -> BVar Double -> Component
entry state value = mkEntryRtrn [textB (currentValue state value)]
                                (setValue state value)
```

1.5.3. Using Radio buttons

The next stage is to define the set of radio buttons that set the current mode. We can do this as follows.

```
radiobuttons :: BVar Convert -> [(String,Convert)] -> Component
radiobuttons state types = do
 r <- mkRadio (Just $ fst $ head types)
 let radio (s,info) =
         mkRadiobutton [text s,useRadio r s]
                       (tellL (input state) info)
 nbeside (map radio types)
```

We can break this down as follows. The set of radio buttons is generated from a list of name and conversion function pairs. When a radio button is pressed it sets the current mode by setting the state `BVar`.

We create a radio object, using `mkRadio`, which all the radiobuttons share. This sets them as a group and guarantees that only one of the buttons may be set at a time. It takes a value to say which radio item should be set initially. (In this case the first element is “Inches”).

```
mkRadio :: Eq a => Maybe a -> GUI (Radio a)
```

We create a radio button using `mkRadiobutton`, which as usual takes a list of configuration information and a listener that it tells about button clicks.

```
mkRadiobutton :: [Conf Radiobutton] -> Listener () -> Component
```

Each radio button displays its own String name, such as Inches (the `text s` configuration option). Its listener sets the state `BVar` with its own conversion functions when pressed. Finally it uses the radio produced earlier, set with `useRadio`.

```
useRadio :: (Has_useRadio w, Eq a) => Radio a -> a -> Conf w
```

Note that `useRadio` also takes a value of type `a`, which is used as the element’s reference. This results in the “Inches” element becoming the current element.

1.5.4. The complete code

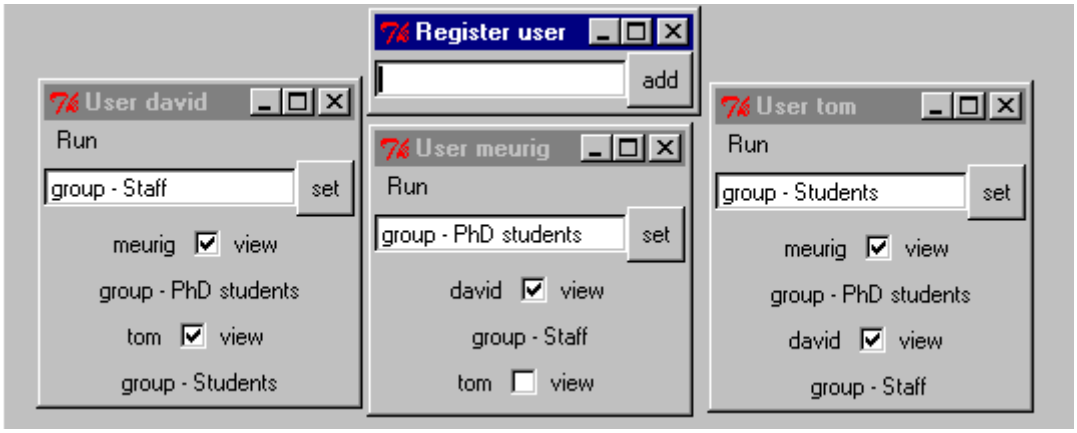
The complete code for the example can therefore be seen below.

```
main :: IO ()
main = display $ mkWindow [title “Converter”] $ do
                     state <- mkBVar (snd $ head units)
                     value <- mkBVar 0
                     converter state value units

type Convert = (Double -> Double,Double -> Double)

units :: [(String,Convert)]
units = [("Inches",(id,id)),
         ("Feet",(\x -> x * 12,\x -> x / 12)),
         ("Yards",(\x -> x * 36, \x -> x / 36)),
         ("Miles", (\x -> x * 63400,\x -> x /63400))]

converter :: BVar Convert -> BVar Double -> [(String,Convert)]
          -> Component
converter state value types =
  radiobuttons state types `above` entry state value


entry :: BVar Convert -> BVar Double -> Component
entry state value =
```

```
        mkEntryRtrn [textB (currentValue state value)]
                            (setValue state value)

currentValue :: BVar Convert -> BVar Double
             -> Behavior String
currentValue state value =
          lift2 getVal (bvarBehavior state) (bvarBehavior value)
  where
    getVal :: Convert -> RealVal -> String
    getVal (toUnit,fromUnit) unit = show $ fromUnit unit

setValue :: BVar Convert -> BVar Double -> Listener String
setValue state value =
    withSnapL setVal (bvarBehavior state) (input value)
  where
    setVal :: String -> Convert -> RealVal
    setVal val (toUnit,fromUnit) = toUnit $ read val

radiobuttons :: BVar Convert -> [(String,Convert)] -> Component
radiobuttons state types = do
 r <- mkRadio (Just $ fst $ head types)
 let radio (s,info) =
          mkRadiobutton [text s,useRadio r s]
                        (tellL (input state) info)
 nbeside (map radio types)
```

1.6. A Dynamic Example

We'll now consider a more dynamic example. Consider the interface below. It allows a group of users to log on to a system and monitor who else is also logged on. Users can log on with the register window. They each have an individual window that displays their name in the title bar. They can change their own details using the entry area. Pressing the "set" button updates their details with the current value of the entry area. Each user can each see the name of the other users and if they wish their details. The "details" field is only shown when the "view" checkbutton is selected.



1.6.1. Defining an individual node

We'll now implement this example. First of all let's start by defining one of the individual nodes that display a user's name and details. The data for each user will be shared, so we'll define a data type `PublicUser` which has a static name and a string behavior representing the dynamic details information for that user.

```
data PublicUser = PublicUser {
    publicName :: String,
```

```
    publicDetails :: Behavior String
    }
```

We can then define how to create a component that displays this information.

```
mkPublicNode :: PublicUser -> Component
mkPublicNode (PublicUser nm details) = do
 visdetails <- mkBVar True
 let name = mkLabel [text nm]
     vischeck = mkCheckbutton [text "view", checkVal True]
                              (input visdetails)
     detailsLbl = mkLabel [textB $ details]
 above (nbeside [name,vischeck])
       (ifB (bvarBehavior visdetails) detailsLbl emptyComponent)
```

A public node consists of a label for the name, a check button and a label for the details. We create a Boolean `BVar` that models the visibility of the details label. The checkbutton then talks to this `BVar`. To make a checkbutton we use `mkCheckbutton`.

```
mkCheckbutton :: [Conf Checkbutton] -> Listener Bool -> Component
```

We set it so that it is in set position initially using `checkVal`.

```
checkVal :: Has_checkVal w => Bool -> Conf w
instance Has_checkVal Checkbutton
```

We can conditionally display a component using `ifB`.

```
class GBehavior w where
  ifB :: GBehavior w => Behavior Bool -> w -> w -> w
instance GBehavior Component
```

When applied to components `ifB b w1 w2` displays component `w1` when `b` is True and `w2` otherwise. (Other members of the `GBehavior` class include `Behaviors` and `Events`.) In this example we display the details label when visdetails is True and an empty component otherwise.

```
emptyComponent :: Component
```

1.6.2. Defining a User's View

Now we define a window for a user. These consist of a button and entry widget to update details, a collection of public nodes and the window itself with a menu that allows a user to exit.

1.6.2.1. The Button and Entry Widget – Introducing Wires

We'll start by defining the button and entry widget. The entry and button together update a String `BVar`, as shown below.

```
mkEntryWithButton :: String -> Listener String -> Component
mkEntryWithButton bname inputL = do
 wire <- mkWire
 let button = mkButton [text bname] (tellL (input wire) ())
     entry = mkEntry [] (event wire) inputL
 beside entry button
```

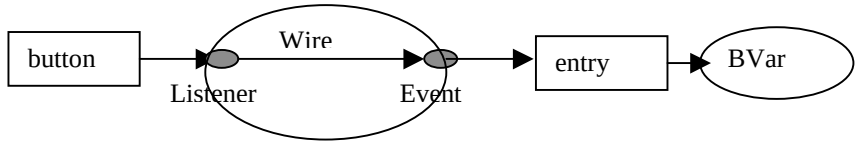
This uses a new concept called a `Wire`. A `Wire` is a limited version of a `BVar`. It has only an input listener and an event. It is therefore a member of the `Has_Event` class.

```
mkWire :: GUI (Wire a)
newWire :: IO (Wire a)

wireInput :: Wire a -> Listener a
wireEvent :: Wire a -> Event a
```

```
instance Has_Event Wire
```

We therefore have the following structure –

- The button talks to the wire when it is clicked
- When the entry hears something on the wire it sends its current value to the BVar



1.6.2.2. Defining the Collection – Introducing Behavioral Collections

Now we'll define the collection of nodes that make up one user's view of those logged on. We do this using a behavioral collection, in this case a list.

```
usersArea :: ListB PublicUser -> Component
usersArea users = nabove $ fmap mkPublicNode users
```

Given a behavioral list of `PublicUsers` we can map the `mkPublicNode` along the list and then place the objects above each other. Note that `nabove` places a collection of components above each other.

```
type ListB a
instance Functor ListB

class PackCollection c w where
 nabove :: c w -> w

instance PackCollection ListB Component
```

When rendered the `ListB` will incrementally update the screen only making necessary changes, rather than redisplaying everything.

1.6.2.3. Making list collections

To make a list collection we use a special type of `BVar`, a `ListBVar`.

```
type ListBVar a

mkListBVar :: Eq a => [a] -> GUI (ListBVar a)
newListBVar :: Eq a => [a] -> IO (ListVar a)
```

When creating a `ListBVar` we give it an initial list of values. Elements in a behavioral list must have equality defined upon them.

For the purposes of this example we will need to append elements to this list and delete them. We can do this using `appendListB` and `deleteListB`.

```
appendListB :: Eq a => ListBVar a -> Listener a
deleteListB :: Eq a => ListBVar a -> Listener a
```

We can extract the `ListB` from the `ListBVar` using collection.

```
collection :: ListBVar a -> ListB a
```

We can also create a `ListB` that always contains just a constant single list using `constantListB`.

```
constantListB :: [a] -> ListB a
```

These behavioral collections are powerful. We can for, instance, apply filter and sort to `ListBs`. For the full range of operations on behavioral collections see Chapter 3.

1.6.2.4. Defining the Window and Menu

We now define an individual user window. Firstly, we define the data for a private user. These have a name, some details information that is held in a `BVar`, they know about a list of other users, and they have an exit listener that is used to log them off.

```
data PrivateUser = PrivateUser {
    name :: String,
    details :: BVar String,
    otherUsers :: ListB PublicUser,
    exit :: Listener ()
    }
```

We define the user window as follows.

```
mkUserWindow :: PrivateUser -> WComponent
mkUserWindow (PrivateUser nm details others exit) = do
 let winmenu = mkMenu [] [mcascade [text "Run"]
                          (mkMenu [] [mbutton [text "Exit"] exit])]
 mkWindow [title $ "User " ++ nm,useMenu winmenu] $
   let editarea = mkEditWithButton "set" (input details)
       usersarea = usersArea others
   in above editarea usersarea
```

This definition can be broken down into several parts. We create a new window component using mkWindow.

```
mkWindow :: [Conf Window] -> Component -> WComponent
```

We give it a specific title using `title`.

```
title :: String -> Conf Window
```

We give it a specific menu using `useMenu`.

```
useMenu :: GUI Menu -> Conf Window
```

We create a menu using `mkMenu`.

```
mkMenu :: [Conf Menu] -> [MenuItem] -> GUI Menu
```

This takes a list of menu items as well as its configuration information and displays that list. (There is also a version of `mkMenu` that displays a dynamic `ListB` of menu elements. See section 4.10.)

In our case we create one cascading menu over the menu bar of the window and place a single menu button item within this cascading menu. We then just tell the exit menu button to talk to the exit listener.

```
mcascade :: [Conf Cascade] –> GUI Menu -> MenuItem
mbutton :: [Conf Cascade] -> Listener () -> MenuItem
```

1.6.3. Creating the Whole Application

As our final stage we need to create the whole application by making the register window, the pile of windows and the application code.

The register window component simply consists of a button and entry within a window.

```
registerWindow :: Listener String -> WComponent
registerWindow add = mkWindow [title “Register user”] $
```

```
                        mkEntryWithButton “add” add
```

The pile of windows simply consists of one window for each user and the registerWindow.

```
windows :: Listener String -> ListB PrivateUser -> WComponent
windows add users = pile [registerWindow add,
                          pile $ fmap mkUserWindow users]
```

Here pile is a class function that displays a collection of components.

```
class Pile c w where
 pile :: c w -> w

instance Pile ListB WComponent
instance Pile [] WComponent
```

Finally we have to create the application code, that is the list of users, and render the list of windows.

```
main :: IO ()
main = start $ do
 privatelst <- mkListBVar []
 windows (addUser privatelst) (collection privatelst)
```

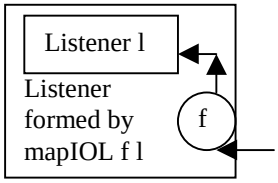
We need to create a user from a String. We can do this using **mapIOL**. This is another member of the family of listener operations and allows us to apply an IO operation to a value before passing it to a listener.

```
mapIOL :: (a -> IO b) -> Listener b -> Listener a
```

Again note the funny looking type. If we think of **Listener** as being of type `a -> IO ()`, then we can think of the definition of mapIOL as follows:

```
mapIOL f l = \x -> do {y <- f x ; l  y}
```

Note that we apply the function f to any values we receive and then pass the value on to l.




```
addUser :: ListBVar PrivateUser -> Listener String
addUser lst = mapIOL mkUser (appendListB lst)
  where
     mkUser :: String -> IO PrivateUser
     mkUser nm = do
          details <- newBVar ""
          let exit = tellL (deleteListB lst) pu
              pu = PrivateUser nm details others exit
              others = fmap toPublic $
                           filterLB (\x -> name x /= nm) $
                                    collection lst
          return pu

     toPublic :: PrivateUser -> PublicUser
     toPublic (PrivateUser {name = nm,details = ds}) =
               PublicUser nm (bvarBehavior ds)
```

To make each user’s public list we filter it so that it does not include an item for themselves and map private user info to public user info. We perform the filtering using filterLB.

```
filterLB :: (a -> Bool) -> ListB a -> ListB a
```

That's it. The complete code for the example is shown again below. A multi-user program in only 70 lines of code.

```
main :: IO ()
main = display $ do
 privatelst <- mkListBVar []
 windows (addUser privatelst) (collection privatelst)

data PublicUser = PublicUser {
    publicName :: String,
    publicDetails :: Behavior String
    }

mkPublicNode :: PublicUser -> Component
mkPublicNode (PublicUser nm details) = do
 visdetails <- mkBVar True
 let name = mkLabel [text nm]
     vischeck = mkCheckbutton [text "view", checkVal True]
                              (input visdetails)
     detailsLbl = mkLabel [textB $ details]
 above (nbeside [name,vischeck])
       (ifB (bvarBehavior visdetails) detailsLbl emptyComponent)

data PrivateUser = PrivateUser {
    name :: String,
    details :: BVar String,
    otherUsers :: ListB PublicUser,
    exit :: Listener ()
    }

instance Eq PrivateUser where
 p == p1 = name p == name p1

mkEntryWithButton :: String -> Listener String -> Component
mkEntryWithButton bname inputL = do
 wire <- mkWire
 let button = mkButton [text bname] (tellL (input wire) ())
     entry = mkEntry [] (event wire) inputL
 beside entry button

usersArea :: ListB PublicUser -> Component
usersArea users = nabove $ fmap mkPublicNode users

mkUserWindow :: PrivateUser -> WComponent
mkUserWindow (PrivateUser nm details others exit) = do
 let winmenu = mkMenu [] [mcascade [text "Run"]
                      (mkMenu [] [mbutton [text "Exit"] exit])]
 mkWindow [title $ "User " ++ nm,useMenu winmenu] $
   let editarea = mkEntryWithButton "set" (input details)
       usersarea = usersArea others
   in above editarea usersarea

registerWindow :: Listener String -> WComponent
registerWindow add = mkWindow [title "Register user"] $
               mkEntryWithButton "add" add

windows :: Listener String -> ListB PrivateUser -> WComponent
windows add users = pile [registerWindow add,
                  pile $ fmap mkUserWindow users]
addUser :: ListBVar PrivateUser -> Listener String
addUser lst = mapIOL mkUser (appendListB lst)
  where
     mkUser :: String -> IO PrivateUser
     mkUser nm = do
          details <- newBVar ""
          let exit = tellL (deleteListB lst) pu
              pu = PrivateUser nm details others exit
              others = fmap toPublic $
                           filterLB (\x -> name x /= nm) $
```

```
                    collection lst
        return pu

    toPublic :: PrivateUser -> PublicUser
    toPublic (PrivateUser {name = nm,details = ds}) =
              PublicUser nm (bvarBehavior ds)
```

1.7. Concepts Summary

We've come across a number of important concepts in this chapter through our set of examples. Here's a summary of the important ones.

1.7.1. The GUI Monad

All widget based GUI actions take place in the GUI monad, which is an extension of the standard IO monad.

1.7.2. Of Behaviors, Events and Listeners

The three basic concepts used in FranTk to manipulate state are `Behaviors`, `Events` and `Listeners`.

A `Behavior` is a continuous value that changes over time. A value of type `Behavior a` is a time varying value of type a. Though it is abstract, it can be thought of as `Behavior a = Time -> a`.

An `Event` is a stream of *occurrences*, each of which has a specific time and value. The type `Event a` denotes an event that generates a value of type `a` when it happens. Though abstract, it can be thought of as `Event a = [(Time,a)]`.

A `Listener` is an abstract type, but it can be thought of as `Listener a = a -> IO ()`. A value of type `Listener a`, is a function, that given a value of type a, performs a side-effecting `IO` action with it. Listeners are therefore consumers of values.

There is an algebra of operations for behaviors, for events and for listeners described in Chapter 2.

1.7.3. Sampling Behaviors

It is often useful to be able to sample the current state on some user input. For instance, to sample interpret some input in terms of the current mode. We can do with using `withSnapL`.

```
withSnapL :: (a -> b -> c) ->Behavior b ->Listener c -> Listener a
```

Note the somewhat back to front type signature that results from listeners being consumers of values. The new listener receives a value, composes it with the current value of the behavior, using the given function, and passes it to the original listener.

1.7.4. Representing State

To represent state in FranTk we use a **BVar**. A value of type **BVar a** represents some abstract mutable state of type a.

```
data BVar a
```

We can create a new `BVar` within the `GUI` or the `IO` monad. Most commonly we will be using them within the `GUI` monad.

```
mkBVar :: a -> GUI (BVar a)
newBVar :: a -> IO (BVar a)
```

It is possible to get a `Behavior` from a `BVar`.

```
bvarBehavior :: BVar a -> Behavior a
```

It is possible to get an `Event` from a `BVar`

```
bvarEvent :: BVar a -> Event a
```

We can set the value of a `BVar` using its `Listener`.

```
bvarInput :: BVar a -> Listener a
bvarUpdInput :: BVar a -> Listener (a -> a)
```

The listener accessed by `bvarInput` updates the `BVar` to its given value. This will alter the value of the BVar's behavior and generate an event occurrence. The listener accessed by `bvarUpdInput` updates the BVar by applying the given function to its current value.

1.7.5. Of Wires

When we only need events and listeners we can use a simpler abstraction called a `Wire`. A `Wire` is a limited version of a `BVar`. It has only an input `Listener` and an `Event`.

```
mkWire :: GUI (Wire a)
newWire :: IO (Wire a)

wireInput :: Wire a -> Listener a
wireEvent :: Wire a -> Event a
```

1.7.6. The Has_Event class

To simplify names a little, `BVar` and `Wire` are both instances of the `Has_Event` class.

```
class Has_Event c where
 input :: c a -> Listener a
 event :: c a -> Event a
```

1.7.7. Of Behavioral Collections

There are collection versions of `BVar` that represent behavioral collections but are more efficient to display as they can be rendered incrementally.

For instance, we can have a `ListBVar` which is a behavioral list collection variable.

```
type ListBVar a

mkListBVar :: Eq a => [a] -> GUI (ListBVar a)
newListBVar :: Eq a => [a] -> IO (ListVar a)
```

There are operations to update these collections specific to their type. For instance a ListBVar contains an operation to append items to the list.

```
appendListB :: Eq a => ListBVar a -> Listener a
```

We can extract the behavioral collection such as ListB from the ListBVar using the function collection.

```
collection :: ListBVar a -> ListB a
```

1.7.8. Of Components and Widgets

All functions that create objects such as buttons and labels produce components.

```
type Component = GUI Widget
```

A `Component` is an action that produces a `Widget`. A `Widget` is an abstract data type representing primitive Tcl widgets. A `Widget` may in fact be made up of several primitive tcl widgets, and may be dynamic changing its appearance over time.

As well as basic components there are several other types including window components. We display basic components by running them inside window components.

```
type WComponent = GUI WindowWidget
withRootWindow :: [Conf Window] -> Component -> WComponent
mkWindow :: [Conf Window] -> Component -> WComponent
```

1.7.9. Configuration Information

To set the appearance of a component we pass in configuration information. This configuration information is typed so that it can only be applied to the correct sort of widget. We use type classes to overload the configuration options to keep names simple. For instance, both a button and a label can take textual configuration information and so are members of the `Has_text` class.

```
class Has_text w
text :: Has_text w => String -> Conf w
instance Has_text Label
mkLabel :: [Conf Label] -> Component
```

To make a component that changes dynamically we use behavioral configuration information. For instance, we can set some changing text using a String behavior.

```
textB :: Has_text w => Behavior String -> Conf w
```

1.7.10. Composing Components

We compose components using operators such as `above` and `beside`.

```
class Packable w where
 above, beside :: w -> w -> w

instance Packable Component
```

We can also compose collections of components.

```
class PackCollection c w where
  nabove,nbeside :: c w -> w

instance PackCollection ListB Component
instance PackCollection [] Component
```

We compose window components using pile.

```
class Pile c w where
 pile :: c w -> w

instance Pile ListB WComponent
instance Pile [] WComponent
```

When rendering a static number of windows, we could just render each individually. The use of piles is, however, vital for rendering a dynamic list of components.

1.7.11. Rendering Components

We render a window component using render.

```
render :: WComponent -> GUI ()
```

Finally, to run the GUI actions that we have produced we use start. This runs the action and then starts up the tcl-tk event loop. This event loop will run until the graphical user interface quits, at which point it will return.

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

Chapter 2 - Dealing with State in FranTk

The previous chapter discussed the basic concepts in FranTk. The most significant feature about FranTk is that all state is modelled in terms of listeners, events and behaviors. In this chapter we'll present the algebra of operators available on each. You probably won't need most of these in day to day life but sometimes they can be very handy.

2.1. What can we really do with Listeners

To summarise again, a `Listener` is an abstract type, but it can be thought of as `Listener a = a -> IO ()`. A value of type `Listener a`, is a function, that given a value of type a, performs a side-effecting IO action with it.

Listeners are therefore consumers of values. This has important concepts for how the listener algebra is structured. The types seem to be reversed as we apply a function to values that a listener is about to receive before passing them to a listener to be consumed.

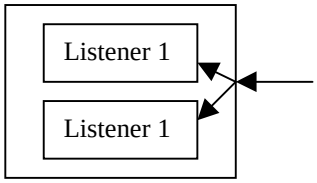
2.1.1. The listener algebra

The first simple listener we have is `neverL`. This is a listener that does nothing with any value it receives.

```
neverL :: Listener a
```

Next we have `mergeL` which merges two listeners to produce a new listener. The combined listener consumes values and passes them to both `l1` and `l2`.

```
mergeL :: Listener a -> Listener a -> Listener a
```



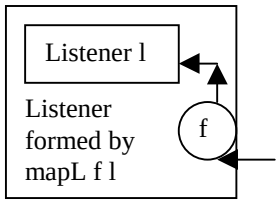
There is also a version of mergeL that takes a list of listeners and merges them all.

```
anyL :: [Listener a] -> Listener a
anyL = foldr mergeL neverL
```

Now we come to mapping functions across listeners. There are two versions of map one for pure functions and one for mapping IO actions across listeners

```
mapL :: (a -> b) -> Listener b -> Listener a
mapIOL :: (a -> IO b) -> Listener b -> Listener a
```

Note the strangely inverted type. This is because listeners are consumers of values not producers. If we think of `Listener` as being of type `a -> IO ()`, then we can think of the definitions of `mapL` and `mapIOL` as follows:




```
mapL f l = \x -> l (f x)
mapIOL f l = \x -> do {y <- f x ; l  y}
```

Note that we apply the function f to any values we receive and then pass the value on to l.

There is a version of `mapIOL` for applying GUI actions to listeners, `mapGUIL`. As there is environment being carried around in the GUI monad this function needs to be applied within the GUI monad.

```
mapGUIL :: (a -> GUI b) -> Listener b -> GUI (Listener a)
```

The next important, and very commonly used function is `tellL`. This takes a listener and a value, and returns a listener that ignores its argument and always performs its action with this value. It is just a special version of `mapL`.

```
tellL :: Listener a -> a -> Listener b
tellL l a = mapL (const a) l
```

The next set of functions filter the values being received by a listener. There is a version of filter, similar to `mapMaybe`, that takes an `(a -> Maybe b)` function, and uses it to filter only values that return `Just b`. Note that `mapMaybeL` takes a listener, l, and makes a new listener that filters the value it receives, before passing them to l.

```
mapMaybeL :: (a -> Maybe b) -> Listener b -> Listener a
```

Built on top of this there is a combinator that behaves similarly to the standard filter function.

```
filterL :: (a -> Bool) -> Listener a -> Listener a
filterL f l = mapMaybeL (\a -> if f a then Just a else Nothing) l
```

There is also a version that makes a listener that hears maybe values but only consumes the `Just` values.

```
isJustL :: Listener a -> Listener (Maybe a)
isJustL = mapMaybeL id
```

We can produce a listener accepting a list of values, and tell it to consume each in turn using `fromListL`.

```
fromListL :: Listener a -> Listener [a]
```

This is defined in terms of a more general mapping function `mapLs` which takes a listener, l, accepting values of type b, and a function, f, producing a list of bs. Every time it hears a value, it applies f, and then tells l to consume each b value in turn.

```
mapLs :: (a -> [b]) -> Listener b -> Listener a
```

We define `fromListL` simply as follows.

```
fromListL = mapLs id
```

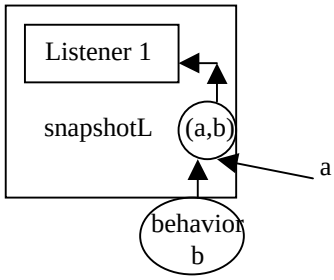
We can create a one shot listener that consumes one value and then behaves as neverL using onceL.

```
onceL :: Listener a -> Listener a
```

Here we see some of the power hidden within the listener abstraction. A listener has the ability to remove itself when it has finished performing useful activity.

We can make a listener snapshot a behavior and consume its current value. For instance, we have `snapshotL`.

```
snapshotL :: Behavior b -> Listener (a,b) -> Listener a
```



Another useful function is withSnapL that also accepts a function to apply to the values from the behavior and the listener.

```
withSnapL :: (a -> b -> c) -> Behavior b -> Listener c
          -> Listener a
withSnapL f bh l = snapshotL bh (mapL (uncurry f) l)
```

These functions are very useful. As all state in FranTk is held in behaviors, and updated through listeners, the ability to snapshot the value of a behavior is the fundamental operation to read the current value of mutable state. (See section 1.5.2 for an example of use.)

We can get access to the time that a listener consumes a value using `withTimeL`.

```
withTimeL :: Listener (a,Time) -> Listener a
```

Here the type Time is a synonym for Double, and represents the number of seconds since the start of the program.

We can make a listener that simply performs an IO action using `mkL`.

```
mkL :: (a -> IO ()) -> Listener a
```

We can also make a listener that performs a GUI action, though this requires a GUI action to produce, to extract the necessary environment information from the monad.

```
mkGUIL :: (a -> GUI ()) -> GUI (Listener a)
```

If we need to fire a listener manually we can achieve this via `fireListener`. This provides the listener via an IO action.

```
fireListener :: Listener a -> IO (a -> IO ())
```

These can be useful if you want to make a listener that performs some explicit IO, such as printing something out for debugging reasons. However, most of the time it is not necessary, as listeners are produced by all of the behavior variables and wires, discussed in chapter 1. In particular, it is important not to deconstruct and then reconstruct listeners using these primitives. This may have undesirable effects. For instance, the listener formed by `remakeL l` will not behave as efficiently as l.

```
remakeL :: Listener a -> IO (Listener a)
remakeL l = do
  act <- fireListener l
  mkL act
```

2.1.2. Primitive Listener Operations

There are a few more primitive listener operations that can be used to define all of the algebra above. These are `liftL1` and `liftL2`. These compose listeners and allow us to apply functions over them. The first `liftL1` takes one listener and produces another by redefining what it does with its action when it consumes a value.

```
liftL1 :: ((a -> IO ()) -> b -> IO ()) -> Listener a -> Listener b
```

This can best be seen by example. For instance, we can define `mapL` in terms of `liftL1`.

```
mapL :: (a -> b) -> Listener b -> Listener a
mapL f = liftL1 $ \act val -> act (f val)
```

The function `liftL2` allows us to merge two listeners and redefine their behavior.

```
liftL2 :: ((a -> IO ()) -> (b -> IO ()) -> Occ c -> IO ())
       -> Listener a -> Listener b -> Listener c
```

We can define `mergeL` using `liftL2`.

```
mergeL :: Listener a -> Listener a -> Listener a
mergeL = liftL2 $ \act1 act2 val -> do act1 val;act2 val
```

2.2. What can we really do with Events

Recall from chapter 1 that an `Event` is a stream of *occurrences*, each of which has a specific time and value. The type `Event a` denotes an event that generates a value of type `a` when it happens.

The algebra of operations available on events resemble closely those available for listeners. The types of the event algebra are mirror images of those in the listener algebra, being structured in the more obvious manner. This is because events are *producers* of values.

2.2.1. Connecting listeners and events

Given an event we can add listeners to that event. These will perform actions on every event occurrence. Listeners and events meet with `addListener`.

```
addListener :: Event a -> Listener a -> IO Remover
type Remover = IO ()
```

The function `addListener` adds a listener to an event. The remove action that is returned will then delete that listener when necessary at a later date. Events therefore serve client listeners.

2.2.2. The Event Algebra

We can make an event that never produces any occurrences using `neverE`.

```
neverE :: Event a
```

We can merge two events using `mergeE`. There is also an infix version of this function `.|.` Here we see the first of a number of infix operators that make up the event algebra.

```
mergeE,(.|.) :: Event a-> Event a -> Event a
```

Again there is a list version of merge for events

```
anyE :: [Event a] -> Event a
anyE = foldr (.|.) neverE
```

We can map functions over events using `mapE`.

```
mapE :: (a -> b) -> Event a -> Event b
```

There are also infix operator versions of `mapE`, as well as a version that ignores the value produced and just uses the new value.

```
(-=>) :: Event a -> b -> Event b
(==>) :: Event a -> (a -> b) -> Event b
```

We can apply a filter function to events in a similar way to listeners.

```
mapMaybeE :: Event a -> (a -> Maybe b) -> Event b

filterE :: Event a -> (a -> Bool) -> Event a
filterE e f = mapMaybeE (\v -> if f v then Just v else Nothing)

filterE_ :: Event a -> (a -> Bool) -> Event ()
filterE_ e f = filterE e f -=> ()
```

We can also take an event of maybe occurrences, and drop all those that are `Nothing`, using `isJustE`.

```
isJustE :: Event (Maybe a) -> Event a
isJustE = mapMaybeE id
```

As with listeners we can convert an event with occurrences that contain lists of values into occurrences for each value.

```
mapEs :: Event a -> (a -> [b]) -> Event b

fromListE :: Event [a] -> Event a
fromListE = mapEs id
```

We can define a one shot event using `onceE`. This generates only one occurrence from its argument event and then behaves like `neverE`.

```
onceE :: Event a -> Event a
```

We can sample the time when an event occurs using `withTimeE`.

```
withTimeE :: Event a -> Event (a, Time)
withTimeE_ :: Event a -> Event Time
```

We can sample behaviors on event occurrences using `snapshotE`. This is one of a range of functions that composes events with behaviors. We'll come across the full range in section 2.3.4 when we go on to discuss behaviors.

```
snapshotE :: Event a -> Behavior b -> Event (a,b)
snapshotE_ :: Event a -> Behavior b -> Event b
```

To aid in debugging there is `traceE` which prints a String on every event occurrence.

```
traceE :: Show a => Event a -> String -> Event a
```

2.2.3. The IO based combinators

There are a number of event combinators that accumulate values over time. These live within the IO monad. We'll see why with our first example `accumE`.

We can accumulate a value via a function valued event using accumE.

```
accumE :: a -> Event (a -> a) -> IO (Event a)
```

The behavior of this function can be seen in the following example.

```
test = do
  wire <- mkWire
  e <- accumE 0 (event wire)
  addListener e (mkL print)
  op <- fireListener (input wire)
  op (+ 1)
  op (+ 1)
```

```
op (+ 1)
```

This makes a wire and attaches a listener to the that prints out every occurrence. We there prepare to fire the listener, and fire it three times. This will produce the output:

1
2
3

Note that it is important that accumE lives within the IO monad so that listeners added at different times will all see the same occurrences. For instance, if we had added the listener after `op` was run for the first time we would have seen the output `2 3` as the event accumulates values from the point when `accumE` is run.

We can ask for only distinct occurrences of an event using `distinctE`.

```
distinctE :: Eq a => Event a -> IO (Event a)
```

We can associate the values from an event stream with the values from a list using `withElemE`.

```
withElemE :: Event a -> [b] -> IO (Event (a,b))
withElemE_ :: Event a -> [b] -> IO (Event b)
withElemE_ e bs = withElemE e bs ==> snd
```

We can generate an event level switcher using `switcherE`.

```
switcherE :: Event a -> Event (Event a) -> IO (Event a)
```

Consider the definition.

```
ev <- switcher e ee
```

The event ev starts out behaving like e. Every time an event appears on ee, ev loses interest in the last event and starts behaving like the new event.

If we don't want to lose interest in the first event we can do this using `manyE`.

```
manyE :: Event a -> Event (Event a) -> IO (Event a)
```

This can be useful when defining composite events. For instance, we can define a double click using `mergeE`. Firstly, we define bindMany. This is like monadic bind for events. Every time ev occurs, `bindManyE` ev f will also gain interest in the event produced by f. (We start out like `neverE` because ev has not yet occurred.)

```
bindMany :: Event a -> (a -> Event b) -> IO (Event b)
bindMany ev f = neverE `manyE` (ev ==> f)
```

We also make use of `alarmE` which given the current time and a wait time delays for that period.

```
alarmE :: Time -> Time -> Event ()
```

We therefore define double click as follows.

```
doubleclick :: Event ()
doubleclick = withTimeE_ click `bindMany`
                \t -> (onceE (alarmE t timeoutval) click)
```

On every click we sample the time. We then wait for one of either the timeout to occur or a second click to occur.

We can define a version of scan on events using `accumE`.

```
scanlE :: (a -> b -> a) -> a -> Event b -> IO (Event a)
scanlE accum a0 e = a0 `accumE` (e ==> flip accum)
```

Finally there is also a cominator to add the previous occurrence to the current occurrence, `withPrevE`.

```
withPrevE :: Event a -> a -> IO (Event (a,a))
e `withPrevE` a0 = scanlE (\(older,old) new -> (old,new))
                          (error "withPrevE: no prev", a0)
                          e

withPrevE_ :: Event a -> a -> IO (Event a)
e `withPrevE_` a0 = e `withPrevE` a0 ==> snd
```

2.3. What can we really do with Behaviors

Finally we come to dealing with behaviors. Recall that a `Behavior` is a continuous value that changes over time. A value of type `Behavior a` is a time varying value of type a, that is a function from `Time -> a.`

There are two simple built in behaviors to start with.

```
time :: Behavior Time
constantB :: a -> Behavior a
```

The `time` behavior is just a simple behavior representing the current time.
The `constantB` behavior is a behavior that always has a single constant value.

To avoid clutter in type signatures involving `Behavior` many types have pre-defined synonyms for their behavioral counterparts. For instance, there is StringB for Behavior String. A full list is available with the type signature summary in the Fran appendix.

2.3.1. Lifted Behaviors

We say a type or function, which has been raised from the domain of ordinary Haskell values to behaviors is "lifted". For example, a function such as

```
(&&) :: Bool -> Bool -> Bool
```

can be promoted to a corresponding function over behaviors:

```
(&&*) :: BoolB -> BoolB -> BoolB
```

The type `BoolB` is a synonym for `Behavior Bool`; most commonly used types have a behavioral synonym defined in FranTk. The name `&&*` arises from a simple naming convention in Fran: lifted operators are appended with a `*` and lifted vars are appended with `B`.

The renaming required by && can sometimes be avoided using type classes. For example, an instance declaration such as the following

```
instance Num a => Num (Behavior a)
```

allows all of the methods in `Num` to be applied directly to behaviors without renaming. Constant types in the class definition cannot be lifted by such a declaration. In the `Num` instance above, the type of `fromInteger` is

```
fromInteger :: Num a => Integer -> (Behavior a)
```

The argument to `fromInteger` is not lifted - only the result. This allows integer constants to be treated as constant behaviors. While `fromInteger` works in the expected way, other class methods cannot be used. In the declaration

```
instance Ord a => Ord (Behavior a)
```

is not useful since it defines operations such as

```
(>) :: Behavior a -> Behavior a -> Bool
```

Unfortunately, FranTk needs a `>` function which returns `Behavior Bool` instead of just `Bool`. The `Eq` and `Ord` classes are not lifted using instance declarations. Rather, each method is individually renamed and lifted. These are the lifting functions: they transform a non-behavioral function into its behavioral counterpart:

```
($*)            :: Behavior (a -> b) -> Behavior a -> Behavior b

lift0           :: a -> Behavior a
lift0           =  constantB

lift1           :: (a -> b) -> Behavior a -> Behavior b
lift1 f b1      = lift0 f $* b1

lift2           :: (a -> b -> c) -> Behavior a -> Behavior b
                -> Behavior c
lift2 f b1 b2   = lift1 f b1 $* b2
...
lift7  ...
```

Using these functions, the definition of `(>*)` is

```
(>*)  = lift2 (>)
```

Many Prelude functions have been lifted in FranTk via overloading. For instance, behaviors are instances of Num, Integral, Fractional, Floating.

2.3.2. Reactive Behaviors

Events are used to build *reactive behaviors* which change course in response to events. Reactive behaviors are defined using the `switcherB` function:

```
switcherB :: Behavior a -> Event (Behavior a) -> IO (Behavior a)
```

It assembles a behavior piecewise from an event and an initial one. For instance, `switcher b e` starts off behaving like b, but changes every time a behavior appears on e.

We can define two very important behavior based functions on top of these.

A `stepper` is a switcher for generating a behavior from constant pieces.

```
stepper :: a -> Event a -> IO (Behavior a)
stepper x0 e = switcherB (constantB x0) (e ==> constantB)
```

We can use `stepAccum` to generate a switcher that starts out behaving as `x0` and is updated by the function occurrences of `change`.

```
stepAccum :: a -> Event (a -> a) -> IO (Behavior a)
stepAccum x0 change = do {e <- accumE x0 change;stepper x0 e}
```

For instance, we can define a counting behavior with stepAccum that counts the number of event occurrences since it was created.

```
countB :: Event () -> IO (Behavior Int)
countB e = 0 `stepAccum` e -=> (+1)
```

The implicit parentheses are around the `-=>` expression, since `` `stepAccum` `` has a lower fixity than `-=>`.

2.3.3. The Reactive classes

There are two important classes here that we use to define a range of operations. The first important class is `switcherM`.

```
class SwitcherM b m where
  switcherM :: a -> Event a -> m a
```

On top of this we have untilB. This behaves as `a` until the next occurrence of `e` after which it behaves as the occurrence value.

```
untilB :: SwitcherM a m => a -> Event a -> m a
untilB a e = switcherM a (onceE e)
```

The function `accumB` accumulates a behavior using `f`.

```
accumB :: SwitchableM bv m => (bv -> b -> bv) -> bv -> Event b
                              -> m bv
accumB f soFar e = switcherM soFar (scanlE f soFar e)
```

The instances of class `SwitcherM` include the following.

```
instance SwitcherM (Behavior a) IO
instance SwitcherM (Behavior a) GUI

instance SwitcherM (Event a) IO
instance SwitcherM (Event a) GUI
```

There are instances of `switcherM` for behavioral collections such as `ListB`.

```
instance SwitcherM (ListB a) IO
instance SwitcherM (ListB a) GUI
```

The other important class is `GBehavior`.

```
class GBehavior w where
  ifB :: BoolB -> w -> w -> w
```

This provides a behavior conditional operation. Its instances include

```
instance GBehavior (Event a)
instance GBehavior (Behavior a)
instance GBehavior Component
instance GBehavior WComponent
```

2.3.4. Turning behaviors into events

We can turn a behavior into an event using `toStream`.

```
toStream :: Behavior a -> IO (Event a)
```

This produces a stream of values every time the behavior changes. If applied to a continuous behavior such as `time :: Behavior Double`, it will generate an occurrence at every clock cycle.

We can define a predicate function in terms of this that generates an event every time a behavior has the value true.

```
predicateB :: Behavior Bool -> IO (Event ())
predicateB b = fmap (flip filterE_ id) $ toStream b
```

2.3.5. Sampling behaviors with events

Finally there are a range of other functions that allow sampling of behaviors with events.

We can create and snapshot a behavior at an event occurrence

```
snapshotF :: Event a -> (a -> Behavior b) -> Event (a,b)
snapshotF_ :: Event a -> (a -> Behavior b) -> Event b
```

There is a generalised version of `ifB` and `snapshotF`.

```
whenSnap :: Event a -> Behavior b -> (a -> b -> Bool) -> Event a
whenSnap e b pred = e ‘snapshot‘ b ‘filterE‘ uncurry pred ==> fst
```

Choosing a behavior or an event from an array based on a behavior.

```
class BehaviorArray b where
    (!*) :: Ix ix => Array ix bv -> Behavior ix -> bv

instance BehaviorArray (Event a)
instance BehaviorArray (Behavior a)
```

2.3.6. Sampling behaviors in the IO monad

Sometimes you may need to sample a behavior from the IO monad. At any time a behavior has a given value. You can therefore get its value using `at` and `getTime`. The latter samples a behavior at a given time, the former returns the current time in seconds. These two are in fact the primitives used by the various `withTime` and `snapshot` functions.

```
getTime :: IO Time
at :: Behavior a -> Time -> IO a
```

It is important to note here that the time is constant in any step. A step begins whenever some user input is handled. A behavior will therefore not change until immediately after the input event that updates it. All updates based on the input will be handled, all behaviors will then change and the display will be updated.

2.4. BVars and Wires

In the first chapter we explained how to create `BVars` and `Listeners`, that provide access to behaviors and events. Recall a BVar is a mutable object with a behavior, listener and an event. A wire is an object with only a listener and an event. The interface for producing a `Wire` is:

```
data Wire a
mkWire :: GUI (Wire a)
newWire :: IO (Wire a)

wireInput :: Wire a -> Listener a
wireEvent :: Wire a -> Event a
```

The interface for producing `BVars` is:

```
data BVar a

newBVar :: a -> IO (BVar a)
mkBVar :: a -> GUI (BVar a)
```

```
bvarBehavior :: BVar a -> Behavior a
bvarEvent :: BVar a -> Event a

bvarInput :: BVar a -> Listener a
bvarUpdInput :: BVar a -> Listener (a -> a)
```

To simplify names a little, BVar and Wire are both instances of the Has_Event class.

```
class Has_Event c where
 input :: c a -> Listener a
 event :: c a -> Event a
```

We are now in fact in a position to understand a possible implementation of BVar. A BVar is made from a wire using `accumE` to accumulate an event value based on a function valued event. We then use stepper to turn the event into a behavior.

```
data BVar a = BVar {
  bvarUpdInput :: Listener (a -> a),
  bvarBehavior :: Behavior a,
  bvarEvent :: Event a
  }

mkBVar :: a -> IO (BVar)
mkBVar a = do
 w <- mkWire
 e <- accumE a (event w)
 b <- stepper a e
 return $ Behavior (input w) b e
```

Chapter 3 - Dealing with Collections

3.1. Behavioral Collections

In chapter 1 we introduced the concept of a behavioral collection. These allow us to model dynamic collections of objects, and treat them as behaviors. They can, however, also be incrementally rendered onto a set of widgets so that only changes are redrawn, not the whole collection.

Currently there are two sorts of behavioral collection available, list and set collections. There is no particular reason why we should be restricted to only these types. Others may appear in the future. The type of a behavioral collection is defined as follows.

```
data CollectionB evop c a
```

The collection is parameterised over its update event, collection type and value. The update event would be of type `evop a`, and the static collection of type `c a`.

3.1.1. The MapG class

Before we begin we should introduce one quick class that proves useful when deailing with collections. The `MapG` class. This is a more general version of `mapM`, that will operate over a given collection and class. There are two initial basic instances for lists and arrays.

```
class MapG c a b m where
  mapG :: (a -> m b) -> c a -> m (c b)

instance Monad m => MapG [] a b m
instance Monad m => MapG (Array I) a b m
```

3.1.2. List Collections

List collections offer the following interface.

```
type ListB a = CollectionB ListOp List a

data ListOp a
data List a
```

We can create a constant `ListB` that will always contain the same list.

```
constantListB :: [a] -> ListB a
```

We can get a `Behavior` from a `ListB`.

```
listBehavior :: ListB a -> Behavior [a]
```

This allows us to treat `ListB` s as normal behaviors when convenient. For instance, we could lift standard list functions and apply them to the list behavior, such as defining a lifted elem function.

```
elemB :: Behavior a -> ListB a  -> BoolB
elemB b ls = lift2 elem b (listBehavior ls)
```

`ListB` is a member of the `Functor` and `MapG` classes, making it easy to map functions along the list.

```
instance Functor ListB
instance MapG ListB a b IO
instance MapG ListB a b GUI
```

Currently we can filter and sort elements in the ListB. These sort the list for all time by applying the sorting functions. There are two versions of each function, a simple one that uses a static function to sort or filter, and a dynamic one.

```
filterLB :: (a -> Bool) -> ListB a -> ListB b
sortLB :: (a -> a -> Ordering) -> ListB -> ListB a
```

The dynamic functions deserve more discussion. Their type signatures are as shown below.

```
filterListB :: (a -> Behavior b) -> Behavior (b -> Bool)
            -> ListB a
            -> ListB b
sortListB :: (a -> Behavior b) -> Behavior (b -> b -> Ordering)
          -> ListB b
          -> ListB b
```

They each take a function to extract a behavior from a list element, and a function valued behavior to sort or filter and apply these. We can understand what's going on here best through an example.

Consider our multiuser logon system from section 1.6. Recall each user had a view showing a widget, displaying the name and details, of every other user. Imagine if we wanted to filter that list of objects based on the user's details.

```
usersArea :: ListB PublicUser -> Component
usersArea users = nabove $ fmap mkPublicNode users

data PublicUser = PublicUser {
    publicName :: String,
    publicDetails :: Behavior String
    }
```

We would need a dynamic or behavior based function to perform the filter. We also need to filter based on the publicDetails field which is a behavior. We therefore redefine usersArea so that it takes a behavioral filter function and filters based on the publicDetails field.

```
usersArea :: Behavior (String -> Bool) -> ListB PublicUser
          -> Component
usersArea isvalid users =
    nabove $ fmap mkPublicNode $
           $ filterListB publicDetails isvalid users
```

We can use the ListB type when piling windows or widgets. We also use this type to make menus, listboxes and text areas displaying dynamic data. These uses will be presented in the next chapter.

3.1.3. Set Collections

The set collection implements a similar interface to `ListB`.

```
type SetB = CollectionB SetOp Set a
data SetOp
data Set
```

We can create a constant set and get a behavior from a set. Note that at present sets are just simple lists.

```
constantSetB :: [a] -> SetB a
setBehavior :: SetB a -> Behavior [a]

instance Functor ListB
instance MapG ListB a b IO
instance MapG ListB a b GUI
```

We can filter a set as with lists.

```
filterSB :: (a -> Bool) -> SetB a -> SetB b
```

```
filterSetB :: (a -> Behavior b) -> Behavior (b -> Bool)
           -> SetB a -> SetB b
```

(NB : This function is not yet implemented, coming soon)

There are a few set specific functions as well. We can form the union, the intersection and the set minus of two behavioral sets.

```
unionSetB :: Eq a => SetB a -> SetB a -> SetB a
intersectSetB :: Eq a => SetB a -> SetB a -> SetB a
minusSetB :: Eq a => SetB a -> SetB a -> SetB a
```

3.2. Collection BVars

We need a way to create behavior collections. As we saw in Chapter 1, for this we use `Collection BVars`. `Collection BVars` are all of the following general type.

```
data CollectionBVar evop c a
```

As with behavioral collections they are parameterised over their update event, collection type and value.

We can extract the behavioral collection from a `CollectionBVar` using the function `collection`.

```
collection :: CollectionBVar evop c a -> CollectionB evop c a
```

3.2.1. The ListBVar Interface

The interface for a `ListBVar` is as follows.

```
type ListBVar a = CollectionBVar ListOp List a
```

We can make a `ListBVar` using `mkListBVar`. Note that this requires equality to be defined on list elements.

```
newListBVar :: Eq a => [a] -> IO (ListBVar a)
mkListBVar :: Eq a => [a] -> GUI (ListBVar a)
```

We can extract the behavior from a ListBVar using `listVBehavior`.

```
listVBehavior :: ListBVar a -> Behavior [a]
listVBehavior s = listBehavior $ collection s
```

All of the updates to this collection occur through the BVar's listeners.

We can add insert elements into a list using `insertListB`. This takes a value and a position to place the element and puts the value at that position in the list.

```
insertListB :: ListBVar a -> Listener (a, PlacePos a)

data PlacePos a = PlaceTop | PlaceBottom | PlaceBefore a
                | PlaceAfter a
```

We can delete elements using `deleteListB`.

```
deleteListB :: ListBVar a -> Listener a
```

We can also move elements around in a list using `moveListB`.

```
moveListB :: ListBVar a -> Listener (a,PlacePos a)
```

We can reset the list to a whole new set of values using `resetListB`.

```
resetListB :: ListBVar a -> Listener [a]
```

There are also convenience functions to `append` and `cons` an element onto a list.

```
appendListB :: ListBVar a -> Listener a
appendListB l = mapL (\v -> (v,PlaceBottom)) $ insertListB l

consListB :: ListBVar a -> Listener a
consListB l = mapL (\v -> (v,PlaceTop)) $ insertListB l
```

A ListBVar can be created more efficiently using `mkListBVar'`.

```
newListBVar' :: (a -> Ident) -> [a] -> IO (ListBVar a)
mkListBVar' :: (a -> Ident) -> [a] -> GUI (ListBVar a)
```

This requires that we can map elements to a unique identfier of type `Ident`. This `Ident` value should be a unique identifier for the object.

```
data Ident
  deriving (Eq,Ord)
```

This can be done with the `Identifiable` class.

```
class Identifiable w where
  identify :: w -> Ident
```

Initial members of the `Identifiable` class include String and Int.

```
instance Identifiable String
instance Identifiable Int
```

3.2.2. The SetBVar interface

The `SetBVar` interface is again similar to the `ListBVar` interface.

```
type SetBVar a = CollectionBVar SetOp Set a
```

We create a `SetBVar` using `mkSetBVar`. We can create one more efficiently when elements are members of the `Identifiable` class.

```
newSetBVar :: Eq a => [a] -> IO (SetBVar a)
mkSetBVar :: Eq a => [a] -> GUI (SetBVar a)

newSetBVar' :: (a -> Ident) -> [a] -> IO (SetBVar a)
mkSetBVar' :: (a -> Ident) -> [a] -> GUI (SetBVar a)
```

We can get a behavior from set `SetBVar` using `setVBehavior`.

```
setVBehavior :: SetBVar a -> Behavior [a]
setVBehavior s = setBehavior $ collection s
```

We can add elements using `insertSetB`, delete elements using `deleteSetB`, and reset to a new set of elements using `resetSetB`.

```
insertSetB :: SetBVar a -> Listener a
deleteSetB :: SetBVar a -> Listener a
resetSetB :: SetBVar a -> Listener [a]
```

That's all you need to know about behavioral collections for now. If you want to write your own, look in CollectionB.hs SetB.hs ListB.hs and CollectionBVar.hs for some hints.

Chapter 4 - Introducing Widgets

4.1. Components and Widgets

As we saw in Chapter 1, in FranTk, all widget creation commands create Components. A Component is an action that produces a Widget. A Widget is an abstract representation of a primitive widget. A Widget may be dynamic. There are several types of Component and Widget, representing top level windows; widgets, such as buttons, that live inside top level windows; and canvas items, that live in canvases. We'll see more on canvases in section 4.12.

There is a generic widget type that represents any widget.

```
data WidgetB a
```

There are a few functions that will work on any abstract component or widget.

We can create an empty component or widget.

```
emptyWidget :: WidgetB a
emptyComponent :: GUI (WidgetB a)
```

All components and widgets are instances of the GBehavior type discussed in section 2.3.3.

```
instance GBehavior (WidgetB a)
instance GBehavior (GUI (WidgetB a))
```

All components and widgets can listen to user input. See section 4.13 for more on listening to input.

```
instance Has_Input (GUI (WidgetB a))
instance Has_Input (WidgetB a)
```

We'll now go on to present the different widgets available.

4.2. Windows

A user interface may contain many windows. A window acts as a container for other widgets. A Window Widget represents a window. A WComponent is an action that produces a `Window Widget`.

```
data WW
type WindowWidget = WidgetB WW
type WComponent = GUI WindowWidget
```

The application has a root window, which we can access through `withRootWindow`. We can also create new windows using `mkWindow`. Windows may use configuration information and contain a component.

```
withRootWindow :: [Conf Window] -> Component -> WComponent
mkWindow :: [Conf Window] -> Component -> WComponent

data Window
```

We can render a collection of `WComponents` using `pile`.

```
class Pile c w where
  pile :: c w -> w

instance Pile ListB WComponent
instance Pile ListB WindowWidget
instance Pile [] WComponent
instance Pile [] WindowWidget
```

We can configure a window to display some title text, or some behavior text. We can set it with a size represented as a 2-D vector. We can also set its position.

```
title :: String -> Conf Window
titleB :: StringB -> Conf Window

winsize :: Vector2 -> Conf Window
winsizeB :: Vector2B -> Conf Window

winposition :: Point2 -> Conf Window
winpositionB :: Point2B -> Conf Window
```

We can give a window a particular menu. See section 4.10 for more on menus.

```
useMenu :: GUI Menu -> Conf Window
```

General configuration options available to a Window are background, borderwidth, cursor, height, width, relief, takefocus, highlightbackground, highlightforeground and highlightthicness.

4.3. Components and Layout

Window widgets contain components. A `Component` is an action that produces a `Widget`.

```
data PW
type Widget = WidgetB PW
type Component = GUI Widget
```

We can compose components with a number of basic layout combinators.

We can pack components above and beside each other using the `Packable` class. This provides for

```
class Packable w where
 above,beside :: w -> w -> w
 expandB :: Behavior Bool -> w -> w
 fillB :: Behavior Fill -> w -> w
 expand :: Bool -> w -> w
 fill :: Fill -> w -> w
 pad :: Pad -> w -> w
 padB :: Behavior Pad -> w -> w
 padI :: Pad -> w -> w
 padIB :: Behavior Pad -> w -> w
 anchor :: Anchor -> w -> w
 anchorB :: Behavior Anchor -> w -> w

data Fill = Fill | FillY | FillXY
  deriving (Eq,Show)

data Pad = PadX Int | PadY Int
  deriving (Eq,Show)

data Anchor = N | S | E | W | NE | NW | SE | SW | C
  deriving (Eq,Show)
```

As can be seen, as well as providing simple above and beside combinators, the packable class also provides a range of other functions. Widgets can be made to fill extra space in the X, Y or in both X and Y, using `fill` and `fillB`. Widgets can be made to expand to take up available space in their parent, using `expand` and `expandB`. Widgets can be anchored to a particular corner using `anchor` and `anchorB`. Finally, widgets can be given internal or external padding with `padI/padIB` and `pad/padB` respectively.

To understand the difference between `expand` and `fill` note the following. Consider the example function, `beside a b`. We will refer to the resulting widget as the parent of a and b. With every widget we can associate an inherited area a widget gets from its parent. The occupied area is actually

used for displaying information, and is always a centered subarea of the inherited one. Initially, the occupied and inherited area, equal the minimal dimensions needed by the widget to display its information. After combination with some other widget, the occupied area of the parent is minimal again. If widget `a` is bigger than widget `b`, the inherited area of `a` will equal its occupied area, and the inherited area of `b` will equal the rest of the occupied area of the parent. The `fill` functions make a widget occupy its inherited area either horizontally or vertically. The `expand` function makes a widget claim from its parent all occupied area that is not inherited by one of the other children.

There are a number of infix combinators that build on these basic functions. These combinators apply the same layout function on both arguments. For example `~-~` places two widgets above each other and aligns them in length, `<|>` places them next to each other, aligned in height; `+` is just a compbination of `|` and `-`. Finally, `*` applies an expand operation on the right and left operand.

```
(<>),(<->),(<|>),(<+>) :: Packable w => w -> w -> w
(~~),(~-~),(~|~),(~+~) :: Packable w => w -> w -> w

a <> b = a `beside` b
a <-> b =  fillX a `beside` fillY b
a <|> b = fillY a `beside` fillY b
a <+> b = fillXY a `beside` fillXY b

a ~~ b = a `above` b
a ~-~ b = fillX a `above` fillX b
a ~|~ b = fillY a `above` fillY b
a ~+~ b = fillXY a `above` fillXY b

fillX,fillY,fillXY,flexible :: Packable w => w -> w
fillX = fill FillX
fillY = fill FillY
fillXY = fill FillXY
flexible = expand True . fillXY
```

To pack a collection of widgets beside or above each other use the `PackCollection` class.

```
class PackCollection c w where
 nabove :: c w -> w
 nbeside :: c w -> w
```

A basic widget and a component are both instances of the `Packable` class.

```
instance Packable Widget
instance Packable Component
```

Lists and dynamic lists (ListB) of widgets and Components can be packed above and beside each other.

```
instance PackCollection [] Widget
instance PackCollection ListB Widget
instance PackCollection [] Component
instance PackCollection ListB Component
```

We can also lay out components in a grid. (NB the grid functions are not yet implemented)

```
grid :: [[GridItem]] -> Component
gridItem :: Behavior [GridBagConstraint] -> Component -> GridItem
```

4.4. Labels and Messages

4.4.1. Labels

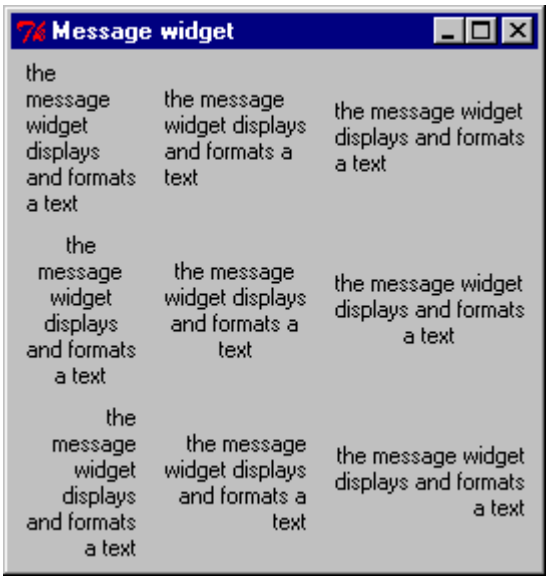
To display information we can use a label. We came across these in section 1.2. A label can display a string or a bitmap. Other valid configuration information includes setting its background color or dimensions.

```
mkLabel :: [Conf Label] -> Component
```

General configuration options available to a Label are anchor, background, bitmap, borderwidth, cursor, font, foreground, height, highlightbackground, highlightcolor, highlightthickness, justify, padx, pady, relief, takefocus, text, underline, width, wraplength.

4.4.2. Messages

Message widgets are similar to labels except that they display structured multi-line strings. A message breaks a long string up into lines. We can use aspect to affect the aspect ratio and justify text left, right or centred. An example message widget is shown below.




```
messageExample :: WComponent
messageExample =
    mkWindow [title "What's the message"] $ matrix 3 $
          [mkMessage [text msg, aspect (75 * i), justify pos]
          | pos <- [LeftJ,CenterJ,RightJ], i  <- [1 .. 3]]
   where
      msg = "the message widget displays and formats a text"
```

We create a message using `mkMessage`.

```
mkMessage :: [Conf Message] -> Component
```

The other configuration options available to a `Message` are anchor, font, highlightthickness, takefocus, background, foreground, padx, text, borderwidth, highlightbackground, pady, cursor, highlightcolor, relief, width.

This example also demonstrates the `matrix` function. This layout function takes a number of columns and a list of components and lays out its arguments in a 2-D matrix.

```
matrix :: Int -> [Component] -> Component
```

4.5. Buttons

In this section we briefly summaries the available range of button widgets.

4.5.1. Command buttons

We came across simple command buttons in section 1.3. When making a button we pass it some configuration information to describe its appearance and a listener to tell about button clicks.

```
mkButton :: [Conf Button] -> Listener () -> Component
```

The configuration options available to a Button are activebackground, activeforeground, anchor, background, bitmap, borderwidth, cursor, font, foreground, height, highlightbackground, highlightcolor, highlightthickness, justify, active_state, padx, pady, relief, takefocus, text, underline, width, wraplength

4.5.2. Check buttons

Checkbuttons have a binary state, True or False. This listener argument is told the current value of the check button when it is selected. `Checkbuttons` were first introduced in section 1.6.1.

```
mkCheckbutton :: [Conf Checkbutton] -> Listener Bool -> Component
```

The state of the checkbutton can be set with `checkVal`, or with a behavioral `checkValB`. (Using the second currently causes the state of the checkbutton to be set every time the value of the behavior changes.)

```
class Has_checkVal w
checkVal :: Has_checkVal w => Bool -> Conf w
checkValB :: Has_checkVal w => Behavior Bool -> Conf w

instance Has_checkVal Checkbutton
```

(Menu checkbuttons are also instances of `Has_checkVal`. See section 4.10.2 for more details.)

The configuration options available to a `Checkbutton` are activebackground, activeforeground, anchor, background, bitmap, borderwidth, cursor, font, foreground, height, highlightbackground, highlightcolor, highlightthickness, indicatoron justify, active_state, padx, pady, relief, selectcolor, takefocus, text, underline, width, wraplength.

4.5.3. Radio buttons

A radiobutton is a member of a group of buttons. Setting one button causes the other buttons to be unset. `Radiobuttons` were first introduced in section 1.5.3.

```
mkRadiobutton :: [Conf Radiobutton] -> Listener () -> Component
```

All radiobuttons in a group share a `Radio`, and so `Radiobutton` is an instance of the `Has_useRadio` class. See section 4.6 for more on the `Radio` type.

The other configuration options available to a Radiobutton are activebackground, activeforeground, anchor, background, bitmap, borderwidth, cursor, font, foreground, height, highlightbackground, highlightcolor, highlightthickness, indicatoron justify, active_state, padx, pady, relief, selectcolor, takefocus, text, underline, width, wraplength.

4.5.4. Making a popup menu button

To make a component button that causes a menu to popup when pressed use a `Menubutton`.

```
mkMenubutton :: [Conf Menubutton] -> Component
```

We can set a menu button to use a given menu using `withMenu` and `withMenuLB`.

```
withMenu :: [Conf Menu] -> [MenuItem] -> Conf Menubutton
withMenuL :: [Conf Menu] -> ListB [MenuItem] -> Conf Menubutton
```

See section 4.10 for more on menus.

The other configuration options available to a Radiobutton are activebackground, activeforeground, anchor, background, bitmap, borderwidth, cursor, font, foreground, height, highlightbackground, highlightcolor, highlightthickness, indicatoron justify, active_state, padx, pady, relief, selectcolor, takefocus, text, underline, width, wraplength.

4.6. The Radio object

We create a radio object, using `mkRadio`, that all radiobuttons in a group share. This sets them as a group and guarantees that only one of the buttons may be set at a time. It takes a maybe value which names the element which should be set initially.

```
mkRadio :: Eq a => Maybe a -> GUI (Radio a)
```

We alter the initial value of the radio using `setRadio`.

```
setRadio :: Eq a => Radio a -> a -> GUI ()
```

We can also set the value of the radio through a listener.

```
radioInput :: Eq a => Radio a -> Listener a
```

The collection of radio objects all share the Radio with `useRadio`.

```
useRadio :: (Has_useRadio w, Eq a) => Radio a -> a -> Conf w
```

Note that `useRadio` also takes a value of type `a`, which it uses as the elements reference.

4.7. Scale Widgets

A scale widget is a widget that allows a user to select a value from a range of values. Scale widgets were first introduced in section 1.2. We create a horizontal or vertical scale using `mkHScale` and `mkVScale` respectively. The listener argument is told the current value of the slider every time it changes.

```
mkHScale :: [Conf Slider] -> Listener Int -> Component
mkVScale :: [Conf Slider] -> Listener Int -> Component
```

To set the value of the scale widget, use `scaleVal`. (Using `scaleValB` currently causes the state of the checkbutton to be set every time the value of the behavior changes.)

```
scaleVal :: Int -> Conf Slider
scaleValB :: IntB -> Conf Slider
```

The other configuration options available to a Radiobutton are activebackground, background, borderwidth, cursor, font, foreground, highlightbackground, highlightcolor, highlightthickness, hor_orient, active_state, relief, sca_from, sca_length, sca_to, takefocus, tickinterval, troughcolor, width, text.

4.8. Listboxes

A listbox is a widget that displays a list of strings, which may be selected.

We can create a listbox using mkListbox.

```
mkListbox :: [Conf Listbox] -> Component
```

We can set the entries in a listbox to be a static string, a behavior list of strings, or a dynamic list (ListB) of strings.

```
listItems :: [String] -> Conf Listbox
```

```
listItemsB :: Behavior [String] -> Conf Listbox

listItemsLB :: ListB String -> Conf Listbox
```
(NB: this is not implemented yet)

(To do: accessing the selected elements of a listbox).

The configuration options available to a listbox are background, foreground, font, borderwidth, cursor, relief, width, hightlightbackground, highlightcolor, highlightthickness, takefocus, height, selectbackground, selectforeground, selectborderwidth, setgrid.

Listboxes are scrollable in X and Y. See section 4.9.

4.9. Scrollbars and scrolling widgets

We create a vertical and horizontal scrollbar using `mkVScrollbar`, and `mkHScrollbar`.

```
mkVScrollbar :: [Conf Scrollbar] -> Component
mkHScrollbar :: [Conf Scrollbar] -> Component
```

Other configuration options available to scrollbars are background, borderwidth, cursor, hor_orient, hightlightbackground, highlightcolor, highlightthickness, relief, troughcolor, takefocus, width.

Scrollbars must be connected to scrollable widgets. We do this using the `Scroll` type.

```
data Scroll
```

We create vertical and horizontal scroll data using `mkVScroll` and `mkHScroll`.

```
mkVScroll :: GUI Scroll
mkHScroll :: GUI Scroll
```

We then make the scrollbar and scrollable widget share the scroll data with `useScroll`.

```
class Has_useScroll w where
 useScroll :: Scroll -> Conf w

instance Has_useScroll Scrollbar
```

Currently scrollable widgets include `Listbox`, `Canvas`, `Entry` and `Edit`.

As an example, we can define a scrollable listbox as follows.

```
mkScrollableListbox :: [Conf Listbox]
                 -> [Conf Scrollbar] -> [Conf Scrollbar]
                 -> Component
mkScrollableEdit cs hs vs = do
 v <- mkVScroll
 h <- mkHScroll
 (mkListbox ([useScroll v,useScroll h] ++ cs)
  `beside`
  (fillY $ mkVScrollbar $ [useScroll v] ++ vs))
  `above`
  (fillX $ mkHScrollbar $ [useScroll h] ++ hs)
```

We first make vertical and horizontal scroll data. We then make a listbox that uses this scroll data. We place this beside a vertical scrollbar, that uses the vertical scroll data, and above a horizontal scrollbar that uses the horizontal scroll data. Note that we make the vertical scrollbar fill available vertical space, and the horizontal scrollbar fill available horizontal space, so that they match the dimensions of the listbox.

4.10. Menus

Menus were first introduced in section 1.6.2.4. We can create a menu that displays either a static list of items, a behavior list of items or a dynamic list (ListB) of menu items.

```
mkMenu :: [Conf Menu] -> [MenuItem] -> GUI Menu
mkMenuB :: [Conf Menu] -> Behavior [MenuItem] -> GUI Menu
mkMenuL :: [Conf Menu] -> ListB MenuItem -> GUI Menu
```

A menu can be made to popup and vanish on event occurrences using popup. A value of Just p, causes the menu to popup at position p. A value of Nothing causes the menu to vanish.

```
popupE :: Event (Maybe Point2) -> Conf Menu
```

The configuration options available to a menu are background, borderwidth, cursor, relief, tearoff.

We will now go on to discuss the possible different menu items.

4.10.1. Menu Item - Button

A menu item button is a simple button that fires a listener when clicked.

```
mbutton :: [Conf Mbutton] -> Listener () -> MenuItem
```

The possible configuration options for an MButton are activebackground, activeforeground, background, bitmap, font, foreground, active_state, underline, text.

4.10.2. Menu Item – Checkbutton

A menu item checkbutton is a button with a binary state. When clicked the buttons tells its listener argument whether the check is selected.

```
mcheckbutton :: [Conf MCheckbutton] -> Listener Bool -> MenuItem
```

The value of the checkbutton can be set with checkVal, using the `Has_checkVal` class. (See section 4.5.2 for more details on this class.)

```
instance Has_checkVal MCheckbutton
```

The other possible configuration options for an MCheckbutton are activebackground, activeforeground, background, bitmap, font, foreground, indicatoron, selectcolor, active_state, underline, text.

4.10.3. MenuItem - Radiobutton

A menu item radiobutton is a button that is a member of a radio group. Only one of the group can be selected at a time.

```
mradiobutton :: [Conf MRadiobutton] -> Listener () -> MenuItem
```

The button can be made a member of a radio group with `useRadio`. See section 4.6 for more details.

The other possible configuration options for an MRadiobutton are activebackground, activeforeground, background, bitmap, font, foreground, indicatoron, selectcolor, active_state, underline, text.

4.10.4. Menu Item - Cascade

We can make a cascading menu with mcascade. This creates a button that displays a given menu when pressed.

```
mcascade :: [Conf MCascade] -> GUI Menu -> MenuItem
```

The possible configuration options available to an MCascade item are activebackground, activeforeground, background, bitmap, font, foreground, underline, active_state, underline, text.

4.10.5. Menu Item – Separator

To add a separator to provide space in a menu use mseparator. A separator accepts no configuration information and has no behavior.

```
mseparator :: MenuItem
```

4.11. Entering Text

There are two types of text entry widgets, entry areas which allow simple single line text entry, and the much more powerful edit areas, which have much of the functionality of a full scale text editor.

4.11.1. Entry Areas

Text entry areas were first introduced in sections 1.5.1 and 1.6.2.1.

We can make a text entry area using `mkEntry'`.

```
mkEntry' :: [Conf Entry] -> Component
```

We can sample the value of a text area using `snapEntry`. Given an event, a listener and a composition function we can set up a sampling function. When a value is heard on the event, the state of the entry is sampled. The value is composed with the String from the entry, using the composition function, and is then told to the listener.

```
snapEntry :: (a -> String -> b) -> Event a -> Listener b
          -> Conf Entry
```

There is a creation function which takes an event and listener, and sends out the state of the entry when a value is heard on the event.

```
mkEntry :: [Conf Entry] -> Event () -> Listener String
        -> Component
mkEntry cs ev l = mkEntry' (snapEntry (\_ s -> s) ev l:cs)
```

There is also a creation function that makes an entry that tells its argument listener the value of the entry, every time the return key is pressed.

```
mkEntryRtrn :: [Conf Entry] -> Listener String -> Component
mkEntryRtrn cs l = do
 w <- mkWire
 keyPress Return (input w) $ mkEntry cs (event w) l

keyPress :: Key -> Listener () -> Component -> Component
```

To do this we create a wire, and bind any `Return` key press to talk to that wire. We then tell the entry to sample the entry every time a value is heard on the wire. More information on listening to events is available in section 4.13.

(To do: sampling and setting the selection of a text entry)

The other configuration options open to entries are background, borderwidth, cursor, font, foreground, highlightbackground, highlightcolor, highlightthickness, justify, password, readOnly, relief, takefocus, width, text.

Entry widgets are scrollable. (See section 4.9)

4.11.2. Edit Areas

Edit areas are much more powerful and allow full scale, multiline text editing. We create an edit widget using mkEdit.

```
mkEdit :: [Conf Edit] -> Component
```

We can sample the value of the edit widet using snapEdit. This operates similarly to snapEntry, discussed in the previous section.

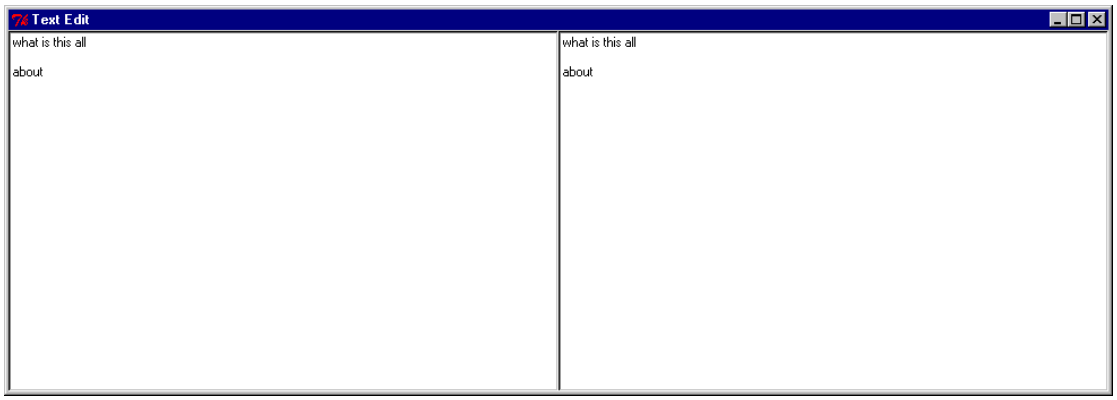
```
snapEdit :: (a -> String -> b) -> Event a -> Listener b
         -> Conf Edit
```

The other general configuration options open to entries are background, borderwidth, cursor, font, foreground, height, highlightbackground, highlightcolor, highlightthickness, padx, pady, readOnly, relief, setgrid, takefocus, width, wrap, text.

Edit widgets are scrollable. (See section 4.9)

4.11.2.1. Sharing state between edit widgets

One of the powerful features provided by Edit widgets is support for writing a shared editor. For instance, below we have two edit widgets side by side. Typing in either alters the text in both.



We can write this text widget with the following code.

```
testShared :: WComponent
testShared = mkWindow [title "Text Edit"] $ do
               st <- mkEditValB emptyEditState
               mkEdit [editValB st,editValL $ editValInput st]
                'above'
                mkEdit [editValL $ editValInput st,editValB st]
```

This can be broken down as follows. Firstly we create an `EditValB`. This is an abstract type representing the state of an edit widget. It can be incrementally updates, and is therefore a form of behavioral collection.

```
data EditValB
```

We can make an edit state object using `mkEditValB` and `newEditValB`.

```
mkEditValB :: EditState -> GUI EditValB
newEditValB :: EditState -> IO EditValB
```

The current state of an editor is represented using `EditState`. This is an abstract type with operations to create an initial edit state containing a String or an empty edit state.

```
data EditState
initEditState :: String -> EditState
```

```
emptyEditState :: EditState
emptyEditState = initEditState ""
```

In our example we have therefore create an initial empty `EditValB` object. We make two edit widgets and tell each to talk to the `EditValB` object.

```
editValInput :: EditValB -> Listener EditVal
editValL :: Listener EditVal -> Conf Edit
```

This causes every use action to be passed on to the `EditValB` object.
(NB: There are a few currently a few bugs with this implementation, with obscure characters. Tcl-Tk does not recognise certain character inputs correctly on some platforms. For instance, with UK keyboards the pound sign may not be recognised properly under NT. There are also some special control shortcuts not handled properly yet, and copy and paste is not handled yet.)

We tell each editor to listen to and display the text of an `EditValB` using `editValB`. The implementation keeps track of where updates originate from so that a given edit widget will not update itself with changes caused by its own user input.

```
editValB :: EditValB -> Conf Edit
```

We can get a String behavior representing the state of the `EditValB` object.

```
editBehavior :: EditValB -> Behavior String
```

(NB: This has not yet been implemented. In particular, the state of the `EditValB` is not fully updated yet. Adding an editor to an `EditValB` that has been passed changes will therefore also currently have unpredictable results.)

From an EditValB object we can also get an event noting every update change.

```
editValEvent :: EditValB -> Event EditVal
```

We can make an edit widget update its view using an `EditVal` event using `editValE`.

```
editValE :: Event EditVal -> Conf Edit
```

We can also set an edit widget with an initial list of EditVal updates, or a behavior list of edit val updates.

```
editVals :: [EditVal] -> Conf Edit
editValsB :: Behavior [EditVal] -> Conf Edit
```

4.11.2.2. What exactly is an Edit update

Updates to an edit state are defined using the `EditVal` data type.

```
data EditVal
```

There are number of functions to construct values of type `EditVal`. We can insert text at a given point (`insertEditVal`), delete text between two given points (`deleteEditVal`), reset the text (`resetEditVal`), or clear it (`clearEditVal`).

```
insertEditVal :: String -> TIndex -> EditVal
deleteEditVal :: TIndex -> TIndex -> EditVal
resetEditVal :: String -> EditVal

clearEditVal :: EditVal
clearEditVal = resetEditVal ""
```

We can insert an `EditMark`, or an `EditTag`. An `EditMark` puts a mark at particular point in some text. An `EditTag` is a way of tagging, and therefore changing the attributes of some section of text. These are discussed in sections 4.11.2.3 and 4.11.2.6 respectively.

```
insertMarkEditVal :: GUI EditMark -> EditVal
insertTagEditVal :: GUI EditTag -> EditVal
```

We can also insert some text at a given point, with an `EditTag` associated with.

```
insertTaggedEditVal :: String -> TIndex -> GUI EditTag -> EditVal
```

Note that a given `EditTag` or `EditMark` can only be added to a single edit widget. This is why we pass in a value of type `GUI EditTag` (or `EditMark`), which is an action that produces an `EditTag` (or `EditMark`).

To define a given point in an edit area we use the `TIndex` type.

```
data TIndex
  deriving (Eq,Show)
```

This has constructor functions to define a point: at a given line number and column number (`tindex`), starting at 1, 1; at the start (`tindexStart`) or end (`tindexEnd`) of the text; at a particular mark (`tindexMark`); at the start (`tindexTagFirst`) or end (`tindexTagLast`) of a tag; or at a given offset from an index (`tindexModMove`).

```
tindex :: Int -> Int -> TIndex
tindexEnd :: TIndex
tindexStart :: TIndex
tindexMark :: Ident -> TIndex
tindexTagFirst :: Ident -> TIndex
tindexTagLast :: Ident -> TIndex
tindexModMove :: TIndex -> ModMove -> TIndex
```

The possible offsets are to the beginning (`LineStart`) or end of the current line (`LineEnd`); beginning (`WordStart`) or end (`WordEnd`) of the current word; moving by a given number of characters (`ModChars`); or moving by a given number of lines (`ModLines`).

```
data ModMove = LineStart | LineEnd | WordStart | WordEnd
             | ModChars Int | ModLines Int
  deriving (Eq,Show)
```

4.11.2.3. Edit Tags

As mentioned in the previous section an `EditTag` can be used to alter the attributes or bind user input to a particular section of text. We can make an edit tag using `mkEditTag`. Some examples of the use of edit tags are given in the following two sections.

```
mkEditTag :: [Conf EditTag] -> GUI EditTag
```

A special form of edit tag is the selection edit tag. This represents the selected area of an edit widget. An edit widget may only have one of these.

```
selectEditTag :: [Conf EditTag] -> GUI EditTag
```

Edit tags are added to an edit widget using the EditVal updates, discussed in section 4.11.2.2.

We can give an `EditTag` a particular unique identifier with `Has_useIdent`.

```
class Has_useIdent w where
  useIdent :: Ident -> Conf w

instance Has_useIdent EditTag
```

It is this unique identifier that is referred to by `TIndex` value (see section 4.11.2.2).

An Edit tag can have input bound to it. It is therefore a member of the Has_Input class. See section 4.13 for more on this.

```
instance Has_Input (GUI EditTag)
```

An EditTag may be set at a given index using the `Has_withIndex` class. It can be given an initial index, or an event stream of indices, or a behavior index. A value of Just t means set the index to t; a value of Nothing means remove the tag from the edit area.

```
class Has_withIndex t w
withIndex :: Has_withIndex w => Maybe t -> Conf w
withIndexE :: Has_withIndex w => Event (Maybe t) -> Conf w
withIndexB :: Has_withIndex w => Behavior (Maybe t) -> Conf w
```

We can sample the indices of an edit tag, from a listener using snapTagL and from an event using snapTagE.

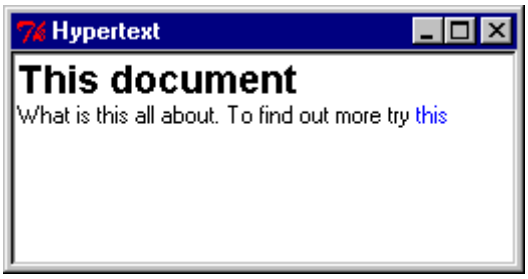
```
snapTagL :: EditTag -> Listener (Maybe (Int,Int),a) -> Listener a
snapTagL_ :: EditTag -> Listener (Maybe (Int,Int)) -> Listener a

snapTagE :: Event a -> EditTag -> Event (a,Maybe (Int,Int))
snapTagE_ :: Event a -> EditTag -> Event (Maybe (Int,Int))
```

The other configuration options to an edit tag available are background, borderwidth, font, foreground, justify, relief, underline, wrap.

We'll see how to use edit tags through two examples now.

4.11.2.4. Edit Tags I - A hypertext example

Consider a simple hypertext system as seen in the figure below.



Hypertext entries are either headings, in bold, 14 point font; hyperlinks to an address; or simple text.

```
type Hypertext = [HypertextTag]
data HypertextTag = Heading String | Text String
                  | Link String Address
type Address = String
```

We can display a hypertext page as follows. We define a hypertext page as displaying a hypertext `Behavior`, that may therefore change over time. When we find an address we pass the address to a listener, which will presumably change the page.

```
hypertextPage :: Behavior Hypertext -> Listener Address
              -> Component
hypertextPage hypertext readPage = do
 mkEdit [editValsB $ lift1 (map (tag readPage)) hypertext,
         readOnly True]
```

```
editValsB :: Behavior [EditVal] -> Conf Edit
```

We make an edit widget that displays the hypertext, using editValsB. Recall from section 4.11.2.1 that editValsB displays a behavior list of EditVals. We make the hypertext page read only. We therefore need to convert hypertext entries into EditVal entries.

```
tag :: Listener Address -> HypertextTag -> EditVal
```

Plain text is results in a String being added to the end of the edit area.

```
tag change (Text s) = insertEditVal s tindexEnd
```

A heading translates into tagged text. We add the String at the end, tagging it with an edit tag, that sets the font *for that bit of text* to 14 point, bold.

```
tag change (Heading s) =
        insertTaggedEditVal (s ++ "\n") tindexEnd
              (mkEditTag [font $ namedFont "Helvetica" 14 [Bold]])
```

A hyperlink also translates into tagged text. In this case we make the tagged text blue, and bind mouse presses with button 1 to tell the change listener about the address.

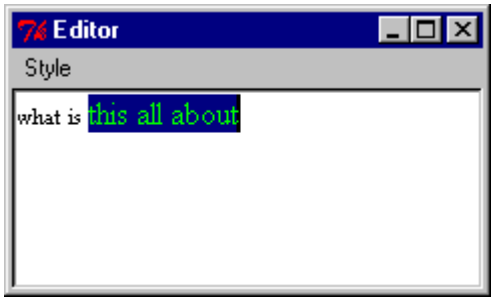
```
tag change (Link s addr) = insertTaggedEditVal s tindexEnd
                              (mousePress 1 (tellL change addr) $
                               mkEditTag [foreground S.blue])

mousePress :: Has_Input w => Int -> Listener () -> w -> w
```

Finally we can make an instance of the hypertext editor. We make a BVar that holds the current page. Pressing a hyperlink will therefore cause the page to be set with the relevant hypertext for the new address.

```
test10 :: WComponent
test10 = mkWindow [title "Hypertext"] $ do
          pagestate <- mkBVar init
          hypertextPage (bvarBehavior pagestate)
                (mapL get $ input pagestate)
 where
  get :: Address -> Hypertext
  get "web:1" = init
  get "web:2" = back

  back, init :: Hypertext

  back = [Heading "Next document",
          Text "Go back to ",
          Link "previous document" "web:1"]

  init = [Heading "This document",
          Text "What is this all about. To find out more try ",
           Link "this" "web:2"]
```

4.11.2.5. Edit Tags II – A more complex text editor

Now consider a more complex text editor. As well as an edit area, users can select an area of text and give it a color, or change its font size.



```
textEditor :: EditValB -> WComponent
textEditor evb = do
  select <- selectEditTag []
  mkWindow [title "Editor",useMenu (textMenu select evb)] $
    mkEdit [font (namedFont "Times" 8 []),
            editValB evb,
            editVals [insertTagEditVal (return select)],
            editValL (editValInput evb)]
```

We make a text editor that uses a shared edit state (`EditValB`). We make the editor display the shared text (`editValB evb`); and talk to it (`editValL (editValInput evb)`). We also create a selection tag, to give access to the current selection, and add it to the edit widget.

We make a menu that uses the selection edit tag. It consists of two radio groups, separated by a menu separator. The first radio group sets the text color, and the second the font size. When a font or color is set a tag is added to the edit state. We make a listener for tag creation using `mkTag`, and make a radio group with `radioGroup`. (See section 4.10 for more on the menu operations.)

```
textMenu :: EditTag -> EditValB -> GUI Menu
textMenu select evb = do
       let tagL :: Listener [Conf EditTag]
           tagL = mkTag select evb
       mkMenu []
           [mcascade [text "Style"]  $ mkMenu [] $
                [radioGroup tagL "Text Color" "black"
                            foregroundB
                            [("black",black),("red",red),
                             ("green",green),("blue",blue)],
                 mseparator,
                 let mkfont :: Int -> Conf EditTag
                     mkfont x = font (namedFont "Times" x [])
                 in radioGroup tagL "Font" "8" mkfont
                       (map (\x -> (show x,x)) [8,10,12,14])
                ]
           ]
```

A `radioGroup` is a cascading menu item, with a group of radio buttons. There is a button for each entry in the vals list; the first item is the name of the entry and the second forms the configuration option for the edit tag. For instance, setting the color sets the foreground of the new tag to the given color. (See section 4.6 for more on the `Radio` type.)

```
radioGroup :: EditTag -> EditValB
           -> String -> String -> (a -> Conf EditTag)
           -> [(String,a)]
           -> MenuItem
radioGroup select evb gpname init config vals =
       mcascade [text gpname] $ do
             rad <- mkRadio (Just init)
             mkMenu [] $
              let mr (c,v) =
                    mradiobutton [text c,useRadio rad c]
                       (mkTag [config v] select evb)
          in map mr vals
```

To make a new tag, we snapshot the indices of the current selection tag (`snapTagL select`). This tells us what area of text the new configuration option will cover. If the selection is empty (`Nothing`), we don't create a tag. If the selection is non-empty, we insert an edit tag, with its initial index equal to the area of the selection. We tell the edit state about new tags (`editValInput evb`). Recall that `mapMaybeL` filters values with a Maybe valued function before passing them to its argument listener.

```
mapMaybeL :: (a -> Maybe b) -> Listener b -> Listener a

mkTag :: EditTag -> EditValB -> Listener [Conf EditTag]
mkTag select evb =
```

```
    snapTagL select (mapMaybeL insertEdit (editValInput evb))
 where
  insertEdit :: ([Conf EditTag],Maybe ((Int,Int),(Int,Int)))
             -> Maybe EditVal
  insertEdit (cs,Nothing) = Nothing
  insertEdit (cs,Just ((x,y),(a,b))) = Just $
   insertTagEditVal
       (mkEditTag (withIndex (Just $ (tindex x y,tindex a b)):cs))
```

4.11.2.6. Edit Marks

An `EditMark` is a mark that can be placed in an edit area. It moves around as the text moves, and so is a way of marking important points in the text such as the beginning of a section. We can create an EditMark using `mkEditMark`.

```
mkEditMark :: [Conf EditMark] -> GUI EditMark
```

The insertion cursor in an edit widget is just a special type of edit mark.

```
insertEditMark :: GUI EditMark
```

As with `EditTags`, we can give an `EditMark` a particular identifier with the `useIdent`; and set its location using `withIndex`. Note that where and `EditTag` has two indices noting its first and last point, a mark has only a single index. (See section 4.11.2.3 for more on useIdent and withIndex).

```
instance Has_useIdent EditMark
instance Has_withIndex TIndex EditMark
```

We can sample the index of an edit tag, from a listener using snapMarkL and from an event using snapMarkE.

```
snapMarkL :: EditMark ->Listener (Maybe (Int,Int),a) -> Listener a
snapMarkL_ :: EditMark -> Listener (Maybe (Int,Int)) -> Listener a

snapMarkE :: Event a -> EditMark -> Event (a,Maybe (Int,Int))
snapMarkE_ :: Event a -> EditMark -> Event (Maybe (Int,Int))
```

4.11.2.7. Copy and Paste

An edit widget will accept copy, cut and paste commands. We can do this using the `clipboardE` configuration option. This accepts an event stream of clipboard actions and makes the edit widget react to these commands.

```
data ClipboardAction = Copy | Cut | Paste

clipboardE :: Event ClipboardAction -> Conf Edit
```

For instance, we can make an edit widget with a menu accepting copy, cut and paste commands. We first create a wire for the menu buttons to talk to, and then make an edit widget that listens to those clipboard commands.

```
textEditor :: IO ()
textEditor = display $ do
  clipB <- mkWire
  let menu =
        mcascade [text "Edit"] $ mkMenu [] $
          [mbutton [text "Copy"] (tellL (input clipB) Copy),
           mbutton [text "Cut"] (tellL (input clipB) Cut),
           mbutton [text "Paste"] (tellL (input clipB) Paste)]
  mkWindow [useMenu menu] $ mkEdit [clipboardE (event clipB)]
```

4.11.2.8. Searching text

An edit widget will also accept search commands. Every time a String appears on an event stream, it searches for a given String, and tells a listener the location of that String. This is coming soon…

```
editFindE :: Event String ->Listener (Maybe ((Int,Int),(Int,Int)))
          -> Conf Edit
```

4.12. Canvas

4.12.1. The Canvas Definition

A `Canvas` is a drawing area that contains a collection of drawing items, such as ovals, or lines, as well standard components such as buttons. To create a canvas we pass in a list of configuration options, and a `CComponent`, which represents the contents of canvas. A canvas example is presented in section 4.12.11.

```
mkCanvas :: [Conf Canvas] -> CComponent -> Component
```

The configuration options available to a canvas are background, borderwidth, cursor, height, highlightbackground, highlightcolor, highlightthickness, relief, scrollregion, takefocus, width.

Canvases are scrollable. (See section 4.9)

4.12.2. The CComponent type

Canvases contain canvas widgets. A value of type `CComponent` is an action that produces a canvas widget.

```
type CanvasWidget = WidgetB CW
type CComponent = GUI CanvasWidget
```

We can place a canvas widget over another using `over`.

```
class Over w where
  over :: w -> w -> w

instance Over CanvasWidget
instance Over CComponent
```

We can stack a pile of canvas objects using pile. The object at the front of the list appears at the top; the object at the end of the list at the bottom. (Recall that we introduced the pile class with Window widgets in section 4.2)

```
class Pile c w where
  pile :: c w -> w

instance Pile ListB CComponent
instance Pile [] CComponent
instance Pile ListB CanvasWidget
instance Pile [] CanvasWidget
```

We can transform canvas items using the `Transformable2B` class.

```
class Transformable2B w where
 (*%) :: Transform2B -> w -> w

instance Transformable2B CComponent
```

This provides operations to move and scale canvas items. (NB: With Tcl they can't be rotated.)

```
move :: Transformable2B bv => Vector2B -> bv -> bv
moveXY :: Transformable2B bv => RealB -> RealB -> bv -> bv
moveTo :: Transformable2B bv => Point2B -> bv -> bv
stretch, shrink :: Transformable2B bv => RealB -> bv -> bv
```

The full set of Fran transformation operations are presented in Fran appendix.

4.12.3. Canvas Item - Ovals

The function `mkCOval`, creates an oval with a size given by the argument vector.

```
mkCOval :: Vector2B -> [Conf COval] -> CComponent
```

The configuration options available to ovals are fillColor, outline, tags, width.

4.12.4. Canvas Items - Lines

The function `mkCLine` creates a line that passes through the list of points provided by the argument list.

```
mkCLine :: [Point2B] –> [Conf CLine] -> CComponent
```

The configuration options available to lines are fillColor, tags, width.

4.12.5. Canvas Items – Arc

The function `mkCArc` creates an arc. It has a radius based on the vector argument. It starts at an angle based on the first real (measured counter-clockwise from the 3 o'clock position); it extends for a number of degrees based on the second argument.

```
mkCArc :: Vector2B -> IntB -> IntB -> [Conf CArc]
       -> CComponent
```

The configuration options available to arcs are fillColor, outline, tags, width.

4.12.6. Canvas Items – Rectangle

The function `mkCRectangle` creates a rectangle, with a size bsed on the argument vector.

```
mkCRectangle :: Vector2B -> [Conf CRectangle] -> CComponent
```

The configuration options available to rectangles are fillColor, outline, tags, width.

4.12.7. Canvas Items – Polygons

The function `mkCPolygon` creates a polygon with corners at all the points in the argument point list.

```
mkCPolygon :: [Point2B] -> [Conf CPoly] -> CComponent
```

The configuration options available to rectangles are fillColor, outline, tags, width.

4.12.8. Canvas Items – Text

The function `mkCText` creates a text item.

```
mkCText :: [Conf CText] -> CComponent
```

The configuration options available to text items are anchor, fillColor, font, justify, tags, text, width.

4.12.9. Canvas Items – Bitmaps

The function mkCBitmap creates a canvas items displaying a bitmap.

```
mkCBitmap :: [Conf CBitmap] -> CComponent
```

The configuration options available to bitmap items are anchor, selectbackground, selectforeground, selectborderwidth, foreground, justify, bitmap, tags.

4.12.10. Canvas Items – Displaying Standard Components

The function `mkCWindow` displays a standard `Component` on a canvas.

```
mkCWindow :: [Conf CWindow] -> Component -> CComponent
```

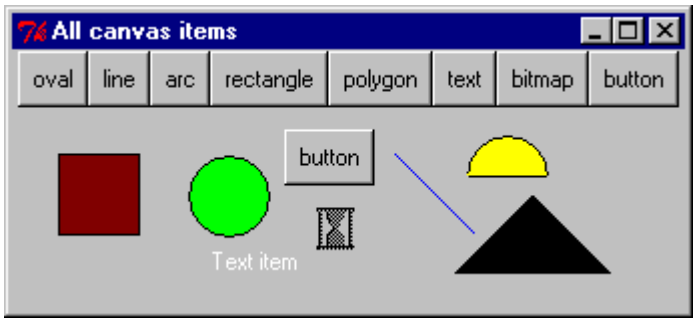
(NB : In Tcl-Tk canvas window items will always appear above (and therefore overlap) other more lightweight canavas items (such as lines or ovals). The stacking order will therefore appear with all canvas widgets stacked appropriately, then all other items stacked appropriately.)

(NB : At the moment reordering of the stacking order of canvas windows is totally broken. They will not be raised or lowered.)

The configuration options available to a canvas window are anchor, justify, tags, width, height.

4.12.11. A Canvas Example

We'll now demonstrate a simple example that shows how to display a dynamic list of canvas items. Items are created using the buttons, moved by dragging with mouse button one, and deleted by pressing mouse button 3.



We make a canvas that displays a given dynamic list. Note that each element of the list contains two parts. A unique identifier (`Ident`) and a widget to display (`CComponent`). This is because list elements must be uniquely identifiable for deletion. We display a pile of items, one for each entry in the list. We listen to mouse movements on the canvas and pass them to each item on the canvas. Each item also has access to the deletion listener, to delete itself from the canvas.

```
canvas :: ListBVar (Ident,CComponent) -> Component
canvas l = do
  mvW <- mkWire
  let mkItem = item (deleteListB l) (event mvW)
  mouseMove (input mvW) $
   mkCanvas [] $ pile $ fmap mkItem (collection l))
```

We create each object using the function`item`. We create two `BVars`, one to record the location of the item, and one to record whether it is moving. When the item is moving we update the *location* `BVar` with any mouse movements (`mvE`). We move the item to the recorded location, and stretch it by a factor of 5. On button 1 mouse presses we start the item moving; on button 1 mouse releases we stop the item moving; and on button 3 presses we delete the item.

```
item :: Listener (Ident,CComponent) -> Event S.Point2
     -> (Ident,CComponent) -> CComponent
item delete mvE obj@(id,c) = do
  p <- mkBVar $ S.point2XY 10 10
```

```
        m <- mkBVar False
        liftIO $ addListener (mvE ‘whenE‘ (bvarBehavior m))
                             (input p)
        moveTo (bvarBehavior p) $ stretch 4 $
          mousePress 1 (tellL (input m) True) $
          mouseRelease 1 (tellL (input m) False) $
          mousePress 3 (tellL delete obj) $
          c
```

We make the set of creation buttons, one for each type of widget.

```
buttonSet :: ListBVar (Ident,CComponent) -> Component
buttonSet = nbeside (map (genButton l) ops)
```

We produce a “create” button with genButton. When the button is pressed it must first generate a unique name for the widget, and then append an entry to the list of items, with that unique name and the relevant `CComponent`. We can generate a unique name using `getUniqueName`.

```
getUniqueName :: GUI Int

genButton :: ListBVar (Ident,CComponent) -> (String,CComponent)
          -> Component
genButton l (nm,w) = do
       let mkact :: a -> GUI (Ident,CComponent)
       mkact = const $ do
          n <- getUniqueName
          return (identify n,w)
       mk <- mapGUIL mkact (appendListB l)
       mkButton [text nm] mk
```

We have one button for each type of widget.

```
ops :: [(String,CComponent)]
ops = zip
   ["oval","line","arc","rectangle","polygon","text","bitmap",
    "button"]
   [mkCOval (vector2XY 10 10) [fillColor S.green],
    mkCLine [point2XY 0 0, point2XY 10 10] [fillColor S.blue],
    mkCArc (vector2XY 10 10) 0 180 [fillColor S.yellow],
    mkCRectangle (vector2XY 10 10) [fillColor S.brown],
    mkCPolygon [point2XY 10 0, point2XY 0 10,point2XY 20 10] [],
    mkCText [text "Text item",anchor NW,fillColor S.white],
    mkCBitmap [bitmap (namedBitmap "hourglass"),anchor NW],
    mkCWindow [anchor NW] $
    mkButton [text "button"] (mkL_ $ print "ouch")
    ]
```

Finally we create a window to put the buttons and canvas in, and the dynamic collection it is to display.

```
runCanvas :: IO ()
runCanvas = display $ mkWidget [title “Canvas”] $ do
   l <- mkListBVar’ fst []
   buttonSet `above` canvas
```

4.13. Listening to user input

We can bind input to any item that is a member of the `Has_Input` class.

```
class Has_Input w
```

It is possible to bind input to any component or edit tag.

```
instance Has_Input (WidgetB a)
instance Has_Input (GUI (WidgetB a))
instance Has_Input (GUI EditTag)
```

There is a range of possible user input we can listen to.

We can listen to mouse button actions with a given button. The Bool refers to whether we are to listen to mouse press or mouse releases. The modifiers restrict what form of input we listen to.

```
mouseButton :: Has_Input w => Int -> Bool -> [Modifier]
            -> Listener () -> w -> w
```

The `Modifier` type is defined as follows. We can restrict input to only repeated events, such as double clicks (`DoubleM`) or triple clicks (`TripleM`); to input only when the control (`ControlM`), alt (`AltM`) or shift (`ShiftM`) key is pressed; to input to when only a button is pressed (`ButtonPressedM`). We can also use system specific modifiers through a String name. This allows access to specific Tcl modifiers that are not covered here.

```
data Modifier =
      DoubleM
    | TripleM
    | ControlM
    | ShiftM
    | AltM
    | ButtonPressedM Int
    | AMod String
 deriving (Show,Eq)
```

We can define standard mouse press and release listeners in terms of `mouseButton`.

```
mousePress :: Has_Input w => Int -> Listener () -> w -> w
mousePress n l = mouseButton n True [] l

mouseRelease :: Has_Input w => Int -> Listener () -> w -> w
mouseRelease n l = mouseButton n False [] l
```

We can listen to mouse movement. Again this may be restricted by a list of modifiers.

```
mouseMove' :: Has_Input w => [Modifier] -> Listener Point2
                          -> w -> w

mouseMove :: Has_Input w => Listener Point2 -> w -> w
mouseMove l = mouseMove' [] l
```

We can listen to mouse enter and leave events, marking entry and exit from a widget. Again these may be restricted with a list of modifiers.

```
mouseEnter' :: Has_Input w => [Modifier] -> Listener () -> w -> w

mouseLeave' :: Has_Input w => [Modifier] -> Listener () -> w -> w

mouseEnter :: Has_Input w => Listener () -> w -> w
mouseEnter = mouseEnter' []

mouseLeave :: Has_Input w => Listener () -> w -> w
mouseLeave = mouseLeave' []
```

We can listen to keyboard input. This may be a press or release (`Bool` argument); we may restrict ourselves to a given key (`Maybe Key` argument); and restrict with a list of modifiers.

```
key :: Has_Input w => Bool -> Maybe Key -> [Modifier]
                   -> Listener Key -> w -> w
```

The possible key values are

```
data Key = Return | Escape | KeyChar Char | Key String | Tab
         | Caps | Shift | Control | Alt | Space | App | BackSpace
         | CursorLeft | CursorDown | CursorRight | CursorUp
```

```
          | Next | Prior | Delete | Insert | Home | End | F Int
  deriving (Eq,Show)
```

For convenience, there are a number of derived keyboard listener functions.

```
keyPress :: Has_Input w => Key -> Listener () -> w -> w
keyPress k l = key True (Just k) [] (tellL l ())

keyRelease :: Has_Input w => Key -> Listener () -> w -> w
keyRelease k l = key False (Just k) [] (tellL l ())

keyPressAny :: Has_Input w => Listener Key -> w -> w
keyPressAny l = key True Nothing [] l

keyReleaseAny :: Has_Input w => Listener Key -> w -> w
keyReleaseAny l = key True Nothing [] l
```

We can listen to resize events on a widget.

```
resizeL :: Has_Input w => Listener Vector2 -> w -> w
```

We can listen to destroy events on a widget. These happen when, for instance, a window is closed.

```
destroyL :: Has_Input w => Listener () -> w -> w
```

4.14. General Confuration Options

There are a range of configuration options that are can be applied to widgets. These are all defined in terms of the `Conf` type.

```
data Conf w
```

We can define generic configuration options using `confGUI`.

```
confGUI :: (w -> GUI ()) -> Conf w
```

There are a range of predefined configuration options that can be applied to objects. As with TkGofer we use type classes to guarantee that only the correct configuration options can be applied to any widget. Along with each static configuration option, there is also a behavior version that has a B suffix on the name. So for instance to set the background of a widget with a behavior color the relevant option would be:

```
backgroundB :: Has_background w => Behavior Color -> Conf w
```

All of these configuration classes require the widget to be a member of the `WidgetItem` class. This class has methods to configure the widget, for instance, to set the color of the widget; to destroy the widget; to get the unique identifier of the widget; to add a finaliser, which is an action that is run when the widget is destroyed.

```
class WidgetItem w where
 cset :: w -> [Config] -> IO () -- configure the widget
 destroy :: w -> IO () -- destroy the widget
 uniqueId :: w -> Ident -- get the unique Identifier of the widget
 addFinaliserW :: w -> IO () -> IO () --
```

Every configurable widget described in this chapter, such as `Button`, is a member of this class.

4.14.1. Setting the color

Set the foreground and background colors of the widget. This uses the color type, defined in the Fran appendix.

```
class WidgetItem w => Has_background w where
  background   :: Color -> Conf w
```

```
class WidgetItem w => Has_foreground w where
  foreground   :: Color -> Conf w
```

Set the foreground and background color of the widget when active.

```
class Has_activebackground w where
  activebackground   :: Color -> Conf w
class Has_activeforeground w where
  activeforeground :: Color -> Conf w
```

Set the background color of the widget when highlighted, and the highlight color.

```
class WidgetItem w => Has_highlightbackground w where
  highlightbackground   :: Color -> Conf w
class WidgetItem w => Has_highlightcolor w where
  highlightcolor   :: Color -> Conf w
```

Set the foreground color when disabled.

```
class WidgetItem w => Has_disabledforeground w where
  disabledforeground :: Color -> Conf w
```

Set the background, foreground and selected color of the selected area of a widget.

```
class WidgetItem w => Has_selectbackground w where
  selectbackground   :: Color -> Conf w
class WidgetItem w => Has_selectcolor w where
  selectcolor   :: Color -> Conf w
class WidgetItem w => Has_selectforeground w where
  selectforeground   :: Color -> Conf w
```

Set the color to fill in a widget

```
class WidgetItem w => Has_fillColor w where
  fillColor   :: Color -> Conf w
```

Set the outline color of a widget.

```
class WidgetItem w => Has_outline w where
  outline   :: Color -> Conf w
```

Some widgets such as scrollbars have trough areas. Set the fill color for this area.

```
class WidgetItem w => Has_troughcolor w where
  troughcolor   :: Color -> Conf w
```

For widgets that can perform word-wrapping, this option specifies the maximum line length.

```
class WidgetItem w => Has_wraplength w where
  wraplength   :: Int -> Conf w
```

4.14.2. Size based configuration options

Specifies a non-negative integer value indicating desired aspect ratio for the text.

```
class WidgetItem w => Has_aspect w where
 aspect :: Int -> Conf w
```

Set the border width.

```
class WidgetItem w => Has_borderwidth w where
  borderwidth   :: Int -> Conf w
```

Set the widget height and width.

```
class WidgetItem w => Has_height w where
  height   :: Int -> Conf w
class WidgetItem w => Has_width w where
  width   :: Int -> Conf w
```

Specify a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus.

```
class WidgetItem w => Has_highlightthickness w where
  highlightthickness   :: Int -> Conf w
```

Specify extra padding to give the widget.

```
class WidgetItem w => Has_padx w where
  padx   :: Int -> Conf w
class WidgetItem w => Has_pady w where
  pady   :: Int -> Conf w
```

For scale widgets specify the starting value, finishing value and length of the scale.

```
class WidgetItem w => Has_sca_from w where
  sca_from   :: Int -> Conf w
class WidgetItem w => Has_sca_to w where
  sca_to   :: Int -> Conf w
class WidgetItem w => Has_sca_length w where
  sca_length   :: Int -> Conf w
```

Specify the scroll region of the widget. The Rect data type is defined in the Fran appendix.

```
class WidgetItem w => Has_scrollregion w where
  scrollregion   :: Rect -> Conf w
```

Specify the width of the border to draw round the widget when selected.

```
class WidgetItem w => Has_selectborderwidth w where
  selectborderwidth   :: Int -> Conf w
```

4.14.3. Miscellanious options

Set the corner to anchor the widget's position from.

```
class WidgetItem w => Has_anchor w where
  anchor   :: Anchor -> Conf w
data Anchor = N | S | E | W | NE | NW | SE | SW | C
 deriving (Eq,Show)
```

Set the bitmap for the widget. Currently we can use bitmaps with a given name. This will be Tcl-Tk dependent.

```
class WidgetItem w => Has_bitmap w where
  bitmap   :: Bitmap -> Conf w
data Bitmap
namedBitmap :: String -> Bitmap
```

Set the cursor when over the widget. Again currently we can use only Tcl-Tk dependent named cursors.

```
class WidgetItem w => Has_cursor w where
  cursor   :: Cursor -> Conf w
data Cursor
namedCursor :: String -> Cursor
```

Do we export the selection from the widget to the clipboard.

```
class WidgetItem w => Has_exportselection w where
  exportselection :: Bool -> Conf w
```

Set another character to be displayed instead of the input one. This is useful with entry fields, where we might want to make a password entry field that displayed only "*".

```
class WidgetItem w => Has_password w where
  password   :: Char -> Conf w
```

Set the select mode for the widget.

```
class WidgetItem w => Has_selectmode w where
  selectmode   :: SelectMode -> Conf w
data SelectMode = SingleMode | BrowseMode | MultipleMode
                | ExtendedMode
 deriving (Show,Eq)
```

Set the font for the widget. We can currently create fonts with a given name, point size and style.

```
class WidgetItem w => Has_font w where
  font   :: Font -> Conf w
data Font
namedFont :: String -> Int -> [FontStyle] -> Font
```

```
data FontStyle = Bold | Italic | Underline | Overstrike | Roman
  deriving Eq
```

Specify whether the widget should be horizontally or vertically oriented.

```
class WidgetItem w => Has_hor_orient w where
  hor_orient   :: Bool -> Conf w
```

Specify an image to display in the widget.

```
class WidgetItem w => Has_image w where
  image :: Image -> Conf w
data Image
namedImage :: String -> Image
```

Should the selected indicator be displayed for radio and check buttons.

```
class WidgetItem w => Has_indicatoron w where
  indicatoron   :: Bool -> Conf w
```

Specify how to justify the widget.

```
class WidgetItem w => Has_justify w where
  justify   :: Justify -> Conf w
data Justify = LeftJ | RightJ | CenterJ
 deriving (Show,Eq)
```

Specify the active state of the widget; is it active, disabled or normal.

```
class WidgetItem w => Has_active_state w where
  active_state   :: ActiveState -> Conf w
data ActiveState = Active | Disabled | Normal
 deriving (Show,Eq)
```

Specify whether the widget is to be read only (text entry widgets).

```
class WidgetItem w => Has_readOnly w where
  readOnly :: Bool -> Conf w
```

Specify the relief to use for the widget.

```
class WidgetItem w => Has_relief w where
  relief   :: Relief -> Conf w
data Relief = Raised | Sunken | Flat | Ridge | Solid | Groove
 deriving (Show,Eq)
```

Specify whether the widget location should resize in valid grid units (such as the size of a character).

```
class WidgetItem w => Has_setgrid w where
  setgrid   :: Bool -> Conf w
```

Specify whether the widget should try to take the input focus when possible.

```
class WidgetItem w => Has_takefocus w where
  takefocus   :: Bool -> Conf w
```

Specify a set of tags, other names that should be associated with the widget.

```
class WidgetItem w => Has_tags w where
  tags   :: [String] -> Conf w
```

Specify the text to display in the widget.

```
class WidgetItem w => Has_text w where
  text   :: String -> Conf w
```

Specify the interval to place ticks on objects such as scale widgets.

```
class WidgetItem w => Has_tickinterval w where
  tickinterval   :: Int -> Conf w
```

Specify which character to underline to provide rapid bindings on objects such as cascading menus. (Pressing the character activates the menu).

```
class WidgetItem w => Has_underline w where
  underline   :: Int -> Conf w
```

Specify how to wrap text in the widget. Do we wrap on to next line at word or character endings automatically.

```
class WidgetItem w => Has_wrap w where
  wrap   :: Wrap -> Conf w
data Wrap = NoWrap | CharWrap | WordWrap
 deriving (Show,Eq)
```

Specify whether to make a menu a tearoff menu.

```
class WidgetItem w => Has_tearoff w where
  tearoff :: Bool -> Conf w
```

Chapter 5 - Using Concurrency

Most of the time the declarative concurrency you can achieve in FranTk simply using behaviors and events is enough. There are, however, times when real pre-emptive concurrency can be helpful. FranTk provides support for using Haskell threads along with the declarative behavior and event model. This interface is new and experimental and so may change. This interface is only currently worth using with GHC as Hugs provides only non-preemptive concurrency.

Consider the following example. We have an interface to a theorem proving tool. This includes a window with a text entry area to develop the proof and a button to run the prover. When we press the prove button we don't want the whole interface to hang. Instead it would be helpful to have the proof computation occur in a different thread allowing the user interface to continue reacting to input.

We can model this by having the main GUI thread fork off a worker thread to perform the computation. The worker thread needs to be able to return its value and update the relevant BVar within the user interface code by firing a listener. The GUI thread can't block waiting for a result from the worker thread and the worker thread shouldn't directly update the listener itself. If this were to happen we'd have to be explicitly careful within the GUI thread about synchronisation issues.

Instead we provide primitives to allow worker threads to communicate through channel variables with the GUI thread.

addCVarListener :: CVar a -> Listener a -> GUI (IO ())
addChanListener :: Chan a -> Listener a -> GUI (IO ())

(NB: these two are not in frantk yet, coming soon)

These allow the listener to wait for input to appear on the channel variables. The CVar version can be used when the worker thread is only to return one value, as with our example above; the Chan version can be used if the worker thread is to return a whole stream of values. Values appearing in a CVar or Channel will be merged with the streams of values that occur from widgets such as buttons, guaranteeing that the simple semantics of the remaining FranTk code are maintained. In particular, this means that after BVars are updated we can be sure that changes to any behaviors will be propogated to the interface widgets before any further updates are made.

Recall that a CVar and a Chan in concurrent haskell have the following interfaces:

A *channel variable* (`CVar`) is a one-element channel:

```
data CVar a
newCVar :: IO (CVar a)
putCVar :: CVar a -> a -> IO ()
getCVar :: CVar a -> IO a
```

A `Channel` is an unbounded channel:

```
data Chan a
newChan         :: IO (Chan a)
putChan         :: Chan a -> a -> IO ()
getChan         :: Chan a -> IO a
dupChan         :: Chan a -> IO (Chan a)
unGetChan       :: Chan a -> a -> IO ()
getChanContents :: Chan a -> IO [a]
```

Note that it is only safe to have one thread, the GUI thread running FranTk GUI code. Other threads should not directly attempt to alter the interface, or the BVars and wires making up the interface model. Instead other threads talk to the GUI thread through this simple interface, updating the behavioral model of the interface data. This restriction is similar to the treatment of the Swing GUI thread and

worker threads in Java. As with Java, actions performed within the GUI thread should be quick to perform. Heavy weight computation should instead be delegated to worker threads.

Note that the GUI thread can communicate with its worker threads by non-blocking means, such as sending requests down a channel.

In our example, we would therefore have code that looked something like the following. Imagine we have a function runProof that takes some proof info and generates a result, but is a heavyweight computation.

```
runProof :: ProofVal -> IO ProofResult
```

The proveComponent is the button that runs the proof. It takes a Behavior modelling the current proof and a listener that the proof result should be sent to, when the proof is complete. We make a listener that produces a worker. Every time it hears a proof val it creates a new CVar and then forks a worker thread to perform the calculation. This ends by telling its result to the CVar. We then add the proof result listener to the CVar.

```
proveComponent :: Behavior ProofVal -> Listener ProofResult
               -> Component
proveComponent proofB proofResult = do
    w <- mkWorker
    mkButton [text “Prove”] (listen w)
  where
    listen :: Listener ProofVal -> Listener ()
    listen worker = snapshotL_ proofB worker

    mkWorker :: GUI (Listener ProofVal)
    mkWorker = mkGUIL $ \ pval -> do
      cvar <- liftIO newCVar
      liftIO $ forkIO $ do
                res <- runProof pval
                putCVar cvar res
      addCVarListener cvar proofResult
      return ()
```

Recall that the function mkGUIL makes a listener that performs a GUI action.

```
mkGUIL :: (a -> GUI ()) -> GUI (Listener a)
```

Chapter 6 - Fran Appendix

There are a number of numeric types defined as part of Fran.

6.1. Numeric Types

The numeric types and functions are available both a static values and as behaviors. Since the same name is generally used for both the static and behavioral version of a function, only the behavioral names are exported by the `Fran` module. If the non-behavioral functions are needed, the convention is to add `import qualified StaticTypes as S` to the program and qualify static names with `S.`, as in `S.origin2`.

6.1.1. Basic Numeric Types

All scalar types are essentially the same in Fran. Synonyms allow type signatures to contain extra descriptive information such as `Fraction` for values between 0 and 1 but no explicit type conversions are required between the various scalar types.

```
type RealVal  = Double
type Length   = RealVal  -- non-negative
type Radians  = RealVal  -- 0 .. 2pi (when generated)
type Fraction = RealVal  -- 0 to 1 (inclusive)
type Scalar   = Double

type Time     = Double
type DTime    = Time     -- Time deltas, i.e., durations

data Point2     -- 2D point
data Vector2    -- 2D vector
data Transform2 -- 2D transformation

data Point3     -- 3D point
data Vector3    -- 3D vector
data Transform3 -- 3D transformation

type RealB       = Behavior RealVal
type FractionB   = Behavior Fraction
type RadiansB    = Behavior Radians
type LengthB     = Behavior Length
type TimeB       = Behavior Time
type IntB        = Behavior Int

type Point2B     = Behavior Point2
type Vector2B    = Behavior Vector2
type Transform2B = Behavior Transform2

type Point3B     = Behavior Point3
type Vector3B    = Behavior Vector3
type Transform3B = Behavior Transform3
```

6.1.2. Points and Vectors

```
origin2             :: Point2B
point2XY            :: RealB -> RealB -> Point2B
point2Polar         :: LengthB  -> RadiansB -> Point2B
point2XYCoords      :: Point2B  -> (RealB,
RealB)
point2PolarCoords   :: Point2B  -> (RealB,
RealB)
distance2           :: Point2B  -> Point2B  -> LengthB
distance2Squared    :: Point2B  -> Point2B  -> LengthB
linearInterpolate2  :: Point2B  -> Point2B  -> RealB -> Point2B
(.+^)               :: Point2B  -> Vector2B -> Point2B
(.-^)               :: Point2B  -> Vector2B -> Point2B
(.-.)               :: Point2B  -> Point2B  -> Vector2B
```

```
origin3             :: Point3B
point3XYZ           :: RealB -> RealB -> RealB -> Point3B
point3XYZCoords     :: Point3B  -> (RealB,
RealB, RealB)
distance3           :: Point3B  -> Point3B  -> LengthB
distance3Squared    :: Point3B  -> Point3B  -> LengthB
linearInterpolate3  :: Point3B  -> Point3B  -> RealB -> Point3B
(.+^#)              :: Point3B  -> Vector3B -> Point3B
(.-^#)              :: Point3B  -> Vector3B -> Point3B
(.-.#)              :: Point3B  -> Point3B  -> Vector3B

xVector2, yVector2 :: Vector2B   -- unit vectors
vector2XY          :: RealB -> RealB -> Vector2B
vector2Polar       :: RealB -> RealB -> Vector2B
vector2XYCoords    :: Vector2B -> (RealB, RealB)
vector2PolarCoords :: Vector2B -> (RealB, RealB)
instance Num Vector2 -- fromInteger, * not allowed

xVector3           :: Vector3B   -- unit vector
yVector3           :: Vector3B   -- unit vector
zVector3           :: Vector3B   -- unit vector
vector3XYZ         :: RealB -> RealB -> RealB -> Vector3B
vector3XYZCoords   :: Vector3B -> (RealB, RealB, RealB)
vector3Spherical   :: RealB -> RealB -> RealB -> Vector3B
vector3PolarCoords :: Vector3B -> (RealB, RealB, RealB)
instance Num Vector3 -- fromInteger, * not allowed/pre>
```

Note that vectors and points have distinct types. You cannot use + to add a point to a vector. Vectors are a member of the Num class while points are not; thus + works with vectors but not points. Although it is in class Num, the * operator cannot be used for vectors.

Read the '.' in the operators above as 'point' and '^' as 'vector'. Thus .+^ means 'point plus vector'.

6.1.3. Vector Spaces

```
zeroVector       :: VectorSpace v => Behavior v
(*^)                :: VectorSpace v => ScalarB -> Behavior v ->
Behavior v
(^/)                :: VectorSpace v => Behavior v -> ScalarB ->
Behavior v
(^+^),(^-^)       :: VectorSpace v => Behavior v -> Behavior v ->
Behavior v
dot                :: VectorSpace v => Behavior v -> Behavior v ->
ScalarB
magnitude        :: VectorSpace v => Behavior v -> ScalarB
magnitudeSquared :: VectorSpace v => Behavior v -> ScalarB
normalize        :: VectorSpace v => Behavior v -> Behavior v

instance VectorSpace Double
instance VectorSpace Float
instance VectorSpace Vector2
instance VectorSpace Vector3
```

6.1.4. Transformations

The types Transformation2B and Transformation3B represent geometric transformation on images, points, or vectors. The basic transformations are translation, rotation, and scaling. Complex transformations are created by composing basic transformations. The class Transformable2 contains 2D transformable objects.

```
class Tranformable2B a where
   (*%)    :: Transform2B -> a -> a   -- Applies a transform
```

These are the operations on 2D transforms:

```
identity2  :: Transform2B
translate2 :: Vector2B -> Transform2B
rotate2    :: RealB -> Transform2B
```

```
compose2   :: Transform2B -> Transform2B -> Transform2B
inverse2   :: Transform2B -> Transform2B
uscale2    :: RealB -> Transform2B  -- only uniform scaling

move :: Transformable2B a => Vector2B -> a -> a
move dp thing = translate2 dp *% thing

moveXY :: Transformable2B a => RealB -> RealB -> a -> a
moveXY dx dy thing = move (vector2XY dx dy) thing

moveTo :: Transformable2B bv => Point2B -> bv -> bv
moveTo p = move (p .-. origin2)

stretch :: RealB -> ImageB -> ImageB
stretch sc = (uscale2 sc *%) -- 1.0 = 180 degrees

turnLeft, turnRight :: Transformable2B a => FractionB -> a -> a
turnLeft frac im = rotate2 (frac * pi) *% im
turnRight frac = turnLeft (-frac)

instance Transformable2B Point2B
instance Transformable2B Vector2B
instance Transformable2B RectB
```

The treatment of 3D is similar.

```
identity3  :: Transform3B
translate3 :: Vector3B -> Transform3B
rotate3    :: Vector3B -> RealB -> Transform3B
scale3     :: Vector3B -> Transform3B
compose3   :: Transform3B -> Transform3B -> Transform3B
uscale3    :: RealB -> Transform3B

class Tranformable3B a where
   (**%)  :: Transform3B -> a -> a

move3 :: Vector3B -> GeometryB -> GeometryB
move3 dp = (translate3 dp **%)

moveXYZ :: RealB -> RealB -> RealB -> GeometryB -> GeometryB
moveXYZ dx dy dz = move3 (vector3XYZ dx dy dz)

moveTo3 :: Point3B -> GeometryB -> GeometryB
moveTo3 p = move3 (p .-.# origin3)

stretch3 :: RealB -> GeometryB -> GeometryB
stretch3 sc = (uscale3 sc **%)

turn3 :: Transformable3B a => Vector3B -> RealB -> a -> a
turn3 axis angle = (rotate3 axis angle **%)
```

A transformation that doubles the size of an object and then rotates it 90 degrees would be `rotate2 (pi/2) ‘compose2‘ uscale2 2.`

Note that the first transform applied is the one on the right, as with Haskell’s function composition operator `(.)`.

6.1.5. Rectangles

We can create and manipulate rectangles.

```
data Rect
type RectB

-- make a rect from a bottom left corner with a given size
mkRect :: Point2B -> Vector2B -> RectB
```

```
rectFromCorners    :: Point2B -> Point2B  -> RectB
rectFromCenterSize :: Point2B -> Vector2B -> RectB
intersectRect      :: RectB   -> RectB    -> RectB
unionRect      :: RectB   -> RectB    -> RectB
rectContains       :: RectB   -> Point2B  -> BoolB
expandRect         :: RealB   -> RectB    -> RectB
overlapRects       :: RectB   -> RectB    -> BoolB
emptyRect          :: RectB
isEmptyRect        :: RectB -> BoolB
rectCenter         :: RectB -> Point2B
rectSize           :: RectB -> Vector2B
```

Increase the rectangles size by a given vector.

```
increaseRect         :: Vector2B -> RectB -> RectB

rectLL, rectUR, rectLR, rectUL :: RectB -> Point2B
rectWidth, rectHeight :: RectB -> RealB
```

6.2. Fran overloaded functions

Many Prelude functions have been lifted in Fran via overloading:

```
(+)         :: Num a => Behavior a -> Behavior a -> Behavior a
(*)         :: Num a => Behavior a -> Behavior a -> Behavior a
negate      :: Num a => Behavior a -> Behavior a
abs         :: Num a => Behavior a -> Behavior a
fromInteger :: Num a => Integer -> Behavior a
fromInt     :: Num a => Int -> Behavior a

quot        :: Integral a => Behavior a -> Behavior a -> Behavior a
rem         :: Integral a => Behavior a -> Behavior a -> Behavior a
div         :: Integral a => Behavior a -> Behavior a -> Behavior a
mod         :: Integral a => Behavior a -> Behavior a -> Behavior a
quotRem     :: Integral a => Behavior a -> Behavior a ->
                             (Behavior a, Behavior a)
divMod      :: Integral a => Behavior a -> Behavior a ->
                             (Behavior a, Behavior a)

fromDouble   :: Fractional a => Double -> Behavior a
fromRational :: Fractional a => Rational -> Behavior a
(/)           :: Fractional a => Behavior a -> Behavior a -> Behavior
a

sin          :: Floating a => Behavior a -> Behavior a
cos          :: Floating a => Behavior a -> Behavior a
tan          :: Floating a => Behavior a -> Behavior a
asin         :: Floating a => Behavior a -> Behavior a
acos         :: Floating a => Behavior a -> Behavior a
atan         :: Floating a => Behavior a -> Behavior a
sinh         :: Floating a => Behavior a -> Behavior a
cosh         :: Floating a => Behavior a -> Behavior a
tanh         :: Floating a => Behavior a -> Behavior a
asinh        :: Floating a => Behavior a -> Behavior a
acosh        :: Floating a => Behavior a -> Behavior a
atanh        :: Floating a => Behavior a -> Behavior a
pi           :: Floating a => Behavior a
exp          :: Floating a => Behavior a -> Behavior a
log          :: Floating a => Behavior a -> Behavior a
sqrt         :: Floating a => Behavior a -> Behavior a
(**)         :: Floating a => Behavior a -> Behavior a -> Behavior a
logBase      :: Floating a => Behavior a -> Behavior a -> Behavior a
```

These operations correspond to functions which cannot be overloaded for behaviors. The convention is to use the `B` suffix for vars and a `*` suffix for ops.

```
fromIntegerB     :: Num a => IntegerB -> Behavior a
```

```
toRationalB      :: Real a => Behavior a -> Behavior Rational
toIntegerB       :: Integral a => Behavior a -> IntegerB
evenB, oddB      :: Integral a => Behavior a -> BoolB
toIntB           :: Integral a => Behavior a -> IntB
properFractionB   :: (RealFrac a, Integral b) => Behavior a ->
Behavior (b,a)
truncateB           :: (RealFrac a, Integral b) => Behavior a ->
Behavior b
roundB               :: (RealFrac a, Integral b) => Behavior a ->
Behavior b
ceilingB            :: (RealFrac a, Integral b) => Behavior a ->
Behavior b
floorB               :: (RealFrac a, Integral b) => Behavior a ->
Behavior b
(^*)             :: (Num a, Integral b) =>
                            Behavior a -> Behavior b -> Behavior a
(^^*)            :: (Fractional a, Integral b) =>
                            Behavior a -> Behavior b -> Behavior a
(==*)            :: Eq a => Behavior a -> Behavior a -> BoolB
(/=*)            :: Eq a => Behavior a -> Behavior a -> BoolB
(<*)             :: Ord a => Behavior a -> Behavior a -> BoolB
(<=*)            :: Ord a => Behavior a -> Behavior a -> BoolB
(>=*)            :: Ord a => Behavior a -> Behavior a -> BoolB
(>*)             :: Ord a => Behavior a -> Behavior a -> BoolB
cond             :: BoolB -> Behavior a -> Behavior a -> Behavior a
notB             :: BoolB -> BoolB
(&&*)            :: BoolB -> BoolB -> BoolB
(||*)            :: BoolB -> BoolB -> BoolB
pairB            :: Behavior a -> Behavior b -> Behavior (a,b)
fstB             :: Behavior (a,b) -> Behavior a
sndB             :: Behavior (a,b) -> Behavior b
pairBSplit       :: Behavior (a,b) -> (Behavior a, Behavior b)
showB            :: (Show a) => Behavior a -> Behavior String
```

A few list-based functions are lifted, although most of the functions in `PreludeList` are not lifted.

```
nilB             :: Behavior [a]
consB            :: Behavior a -> Behavior [b] -> Behavior [b]
headB            :: Behavior [a] -> Behavior a
tailB            :: Behavior [a] -> Behavior [a]
nullB            :: Behavior [a] -> BoolB
(!!*)            :: Behavior [a] -> IntB -> Behavior a
-- Turn a list of behaviors into a behavior over list
bListToListB :: [Behavior a] -> Behavior [a]
bListToListB = foldr consB nilB

-- Lift a function over lists into a function over behavior lists
liftL :: ([a] -> b) -> ([Behavior a] -> Behavior b)
liftL f bs = lift1 f (bListToListB bs)
```