FranTk – A Declarative GUI Language for Haskell

Meurig Sage
University of Glasgow
Dept of Computing Science
Glasgow, United Kingdom
+44 141 339 8844 ext 0918
meurig@dcs.gla.ac.uk

ABSTRACT

FranTk is a new high level library for programming Graphical User Interfaces (GUIs) in Haskell. It is based on Fran (Functional Reactive Animation), and uses the notions of *Behaviors* and *Events* to structure code. Behaviors are time-varying, reactive values. They can be used to represent the state of an application. Events are streams of values that occur at discrete points in time. They can be used, for instance, to represent user input. FranTk allows user interfaces to be structured in a more declarative manner than has been possible with previous functional GUI libraries. We demonstrate, through a series of examples, how this is achieved, and why it is important. These examples are elements of a prototype, Air Traffic Control simulator. FranTk uses a binding to the popular Tcl/Tk toolkit to provide a powerful set of platform independent set of widgets. It has been released as a Haskell library that runs under Hugs and GHC.

Keywords

Graphical User Interfaces, Haskell, Functional Reactive Programming.

1. INTRODUCTION

Over the last few years there has been a great deal of interest in the development of functional GUI libraries. These have used a number of different mechanisms to allow programmers to structure their code. Some such as TkGofer [14] have used traditional callback based approaches. These make it difficult to structure complex interactive systems as the structure of the application is turned inside out [2]. The application cannot call the GUI library, instead the application must be called by the library. A more popular solution has been the use of imperative concurrency [4]. This style allows an application to be structured as a number of threads that execute concurrently and consume user input. However, these approaches require a programmer to handle the intricacies of full concurrent programming, dealing with mutual exclusion and race conditions between processes. These approaches all force a programmer to use a very imperative style of programming; an unfortunate requirement in a declarative language.

In contrast, the User Interface Management Community has been investigating the use of declarative, "constraint" based approaches for programming interactive systems [5]. Here an application is defined as having a behaviour, responding to user input and updating its state. The appearance of the interface is defined as a function of the application's state. One of the most well-known of these systems, Garnet [8], was implemented in Lisp. However, it still relied on side-effects to implement changes in the interface.

This approach supports *application/interface separation*, where one model of the application is maintained, allowing multiple views of the same state to be easily defined. Support for application/interface separation is provided by many modern programming languages, such as Java, which embraces the Model-View-Controller paradigm in its Swing toolkit (see Section 9). The need for this sort of support can be seen in many modern systems, from alternative views of a set of files in a *Microsoft Windows* folder, to text editors such as *Emacs* that allow the same file to be edited by multiple windows. Use of a declarative style greatly benefits application/interface separation. A programmer can define a single function stating how the interface should display the application state, rather than being forced to state how it should be modified with every change to the application. A declarative definition can therefore be more succinct and easier to read, making it easier to ensure that the interface provides a correct representation of the application.

The difference between those languages developed in the functional community and the User Interface Management Community has been described as being "Declarative in the Small versus Declarative in The Large" [5]. The languages developed in the functional programming community have been declarative in the small, allowing individual aspects of a system to be written in a purely functional style. However, they have been imperative in the large, forcing programmers to structure their programs as a set of imperative actions. In contrast, the user interface management community has concentrated on providing systems that are declarative in the large, allowing user interfaces to be structured as a set of constraints or functions. However, being based largely in imperative languages they have not provided the advantages of higher order functions, and referential transparency when building individual components.

There have been a few notable attempts to overcome these difficulties and combine the advantages of both. These include Fudgets[1], Clock [5] and Pidgets [12] (see Section 9). One other important language seems to lend itself to this style of programming. Fran [3] (Functional Reactive Animation) is a language developed for constructing interactive animations. It uses a high-level modelling approach which allows programmers to describe what an animation should look like, not how it should be implemented. The key notions are the use of *behaviors* and *events*. *Behaviors* are time-varying, reactive values, while *events* are streams of values that occur over time.

This paper presents FranTk, an extension of Fran to take it from graphics programming to the construction of complex user interfaces. The major new contributions in FranTk are as follows:

- FranTk lifts Fran's behaviors and events to widgets. This is the key to the declarative style of programming. The appearance of a widget can be defined *for all time* in terms of FranTk combinators (Section 3). This declarative style is supported when programming both static and dynamic interfaces (Section 7,8). The construction of systems with dynamically changing number of components can be difficult in many GUI systems, and frequently requires a very imperative and sometimes cumbersome style of programming.
- FranTk extends Fran with support for hierarchical interactive displays, allowing access to input from individual components rather from one monolithic window. (Section 3)
- FranTk separates visual composition from semantic wiring. (Section 3.3). These two concepts are fundamental to GUI programming. The first involves geometric composition. For instance, placing one widget above another. The second involves connecting user input from a widget to the application code. This separation is made possible by the introduction of *listeners*, consumers that respond to user input (Section 4). FranTk provides an algebra to compose these listeners in a functional style (Section 5).
- FranTk provides a more efficient implementation of the core Fran combinators. This implementation improves on Fran's data-driven model and use of weak references (Section 10).

2. FRAN – AN INTRODUCTION

Fran is a high level language for constructing reactive animations.

2.1 Fran benefits

There are five key aspects that makes it a good candidate for forming the basis of a User Interface development language.

1. Behavioral Modelling.

Fran uses first-class behavior values to model changing values in an animation. A behavior value is a value that changes over time. It can be thought of as `type Behavior a = Time -> a`.

As an example we can make a circle that follows the wave path shown in figure 1. Its position is a function of time; it moves along the screen as time passes, and up and down as the *sin* of time.

Figure 1 – A wave motion ball

```
moveXY time (sin time) circle
```

```
moveXY :: Behavior Double-> Behavior Double
       -> ImageB -> ImageB
```

```
time :: Behavior Double
```

2. Event Modelling.

Events, like behaviors, are first-class values. An event is a stream of values that occurs at discrete points in time. It can be thought of as `type Event a = [(Time,a)].`

We can use events to model happening in the real world, such as button presses, and predicates based on behavior values, such as collisions between objects. For instance, a left button press is simply `lbp u`, where u is a User argument. An event that occurs once, when the time is greater than 5, is `onceE (predicate (time > 5)).` They can be combined as `lbp u .|. onceE (predicate (time > 5)).` The type signatures for the functions we have used are shown below.

```
lbp ::User -> Event ()
predicate :: Behavior Bool -> Event ()
onceE :: Event a -> Event a
(.!.) :: Event a -> Event a -> Event a
```

3. Declarative Reactivity.

Much of the power of Fran comes from the interaction of behaviors and events. We can define "reactive behaviors", that change as events occur. We can therefore give a declarative rather than an imperative semantics to state changes [3]. For instance, we can describe a color-valued behavior that starts out blue, and then changes to red when the left button is pressed, and changes to green when the right button is pressed.

```
blue `stepper` lbp u -=> red
               .|. rbp u -=> green
```

```
stepper :: a -> Event a -> Behavior a
```

4. Declarative Composition.

Animations can be constructed compositionally. We can create two balls following wave motions, one that moves as the *sin* and one as the *cos* of time. Note that these two animations evolve concurrently, and yet are described in a simple, deterministic manner.

```
moveXY time (sin time) circle
 `over` moveXY time (cos time) circle
```

5. Models and Views.

We can define a behavior to represent the state of an animation and then define a function that transforms this state into an image. For instance, we can describe a moving object abstractly in terms of a data type with behaviors. It has a colour and a location.

```
data Obj = Obj (Behavior Color)
               (Behavior Point2)
```

We can then define a function that turns this colour and location into an image.

```
view (Obj pos col) =
  move pos (withColor col circle)
```

```
move :: Behavior Point2 -> ImageB -> ImageB
withColor :: Behavior Color->ImageB->ImageB
```

The features discussed above provide a powerful approach to building interactive systems. We could describe the state of an interactive system as a behavior, reacting to user input events. These components could be easily composed in a declarative manner. We could also support a Model-View style of programming where the state of the application is described abstractly and the appearance described as a function of that state.

2.2 Fran problems

Fran as it exists in its basic form has two serious conceptual restrictions that must be overcome to extend it to interactive systems design.

1. **Hierarchical input.** The most significant problem with Fran is that it does not provide any support for constructing hierarchical interactive displays. All user input is accessed through the *User* argument passed to the animation. This represents input at the level of the entire graphics window. There is no way to access interaction from only a single component, such as an individual button on the screen. The only conceivable mechanism would be to use global mouse coordinates and calculate whether they were within the bounding box of a given object. This approach does not lend itself at all to building hierarchical collections of components, each with their own coordinate systems. This is not usually a major problem in animations; however, the notion of individual interaction objects is critical to the development of standard user interfaces.
2. **Dynamic collections.** Fran provides behavioral values. These could be used to represent behavior collections of objects. For instance, we could display a dynamic list of objects. However if we were to render such a behavior collection, each time an element were to be added *the entire collection would have to be redrawn.* This would be prohibitively expensive if we needed to continually recreate complex compound collections. Fran requires some notion of a incremental behavioral collection that could be both viewed as a behavior and efficiently and incrementally rendered.

In developing FranTk we have overcome both these problems. We demonstrate through a series of examples how this was done. These examples are elements of a prototype Air Traffic Control (ATC) simulator. After presenting the individual elements, we will discuss (in Section 9), how well FranTk scaled to handling the complete application.

3. A SIMPLE TIMER

We begin with a simple example to introduce some basic concepts. The ATC system contains a timer showing the time, in seconds, that have elapsed since the start of the simulation (see Figure 2). The concepts necessary to produce this interface will be presented in this section.

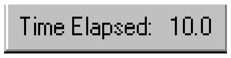

Figure 2 – A simple example

3.1 Introducing Components

The basic conceptual notion in FranTk for handling interaction objects is the *Component*. For instance, in our example we have two label components displayed one beside the other. They appear in a window component.

A `Component` is an action that produces a `WidgetB`. The action will be performed once, when the widget is to be displayed.

```
type Component = GUI WidgetB
```

This definition 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 `WidgetB` is an abstract data type representing a *Widget Behavior*. It is similar to Fran's `ImageB` type representing *Image Behaviors*. An *Image Behavior* models both geometric composition of different images and temporal compositions, as the image changes over time. In the same way a *Widget Behavior* represents both geometric compositions of different widgets and temporal compositions, as the appearance may also be dynamic. As with Fran's `ImageB`, defining `WidgetB` as an abstract data type allows for an efficient implementation.

These basic components can be composed geometrically using simple, functional combinators. For instance, our example is made of two labels, the `title` label and the `timevalue` label. We can place them using `beside`.

```
timer :: Component
timer = title `beside` timevalue
```

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

Though components represent imperative actions that will each produce a widget, we can treat them as an abstract value and so compose them declaratively. This satisfies our aim of supporting a compositional style of programming in FranTk.

FranTk distinguishes between three different types of user interface component, based on how they can be displayed. We have basic components (`Component`), such as labels and buttons, which can be composed using combinators such as `beside`. Secondly, we have graphical components (`CComponent`), such as lines and circles, which can be placed in canvases, and manipulated using animation combinators, based on those provided in Fran. Finally, we have toplevel window components (`WComponent`), which will contain basic components. These can be composed by piling them into groups and can be placed at different locations on the screen.

For the purposes of graphical and temporal composition, there is no difference between a button and a label. They can both be placed in the same way, and so need only be represented by a single component type. Many previous systems have used different types to distinguish between them. This typing was required to allow access to user input from the component, and to apply changes to the component, such as resetting the label, later on. This approach makes it more difficult to geometrically compose a list of widgets as they must be transformed to an untyped display object first. As we will see in the next section, in FranTk all the information necessary to define a component is passed in as a set of parameters so this sort of extra transformation can be avoided. This approach is very significant and is discussed further in Section 4.3.

3.2 Configuring Components

In our example we have two labels with different appearances. One has a static appearance; one a dynamic appearance. We create a label using the `mkLabel` function and configure it using the `text` and `textB` functions.

```
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 [14] 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.

We therefore define the static title label as follows.

```
title :: Component
title = mkLabel [text "Time Elapsed:"]
```

In FranTk, we extend basic configuration with dynamic configuration options. Instead of taking a static value a widget can be given a dynamic behavior value. This approach is the key to the declarative nature of FranTk. Rather than having to carry out imperative updates to change a component's appearance, we can define, using a behavior, what it will look like *for all time*. Given a behavioral model of an application, we can therefore define the appearance of the interface simply as a function of the application's state.

We can define the timer label as follows.

```
timevalue :: Component
timevalue = do
 time <- timeTick 1
 mkLabel [textB (lift1 show time)]
```

```
textB ::Has_text w=>Behavior String->Conf w
timeTick :: Time -> GUI (Behavior Time)
lift1 :: (a -> b) ->Behavior a->Behavior b
```

The function `timeTick` creates a behavior that represents the time, and changes at a given frequency. In our example, the time value changes every second. The function `lift1` maps a function over a behavior, to yield a new behavior. This shows the benefit of the GUI representation of a Component. We can create some local state for a component while still thinking of it as a value.

3.3 Rendering Components

When we have defined an interface in terms of a Window Component, to render it onto the screen we use `render`. This will perform the action necessary to produce the window widget behavior, and then display it.

```
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
```

To test the timer component on its own, we would place it in a window, and so define main as follows.

```
main :: IO ()
main = display (mkWindow [] timer)

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

4. AN INTERACTIVE EXAMPLE

So far we've dealt with an interface with a dynamic appearance but with no interaction. This section demonstrates how interaction is handled in FranTk.

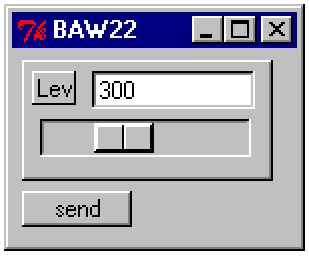

Figure 3 – An interactive example

Consider the interface shown in Figure 3. It shows a simplified version of the tactical data entry widget from the ATC system, which allows the controller to tell the aircraft to change flight level. The proposed level can each be altered, by either typing a value or using the slider. If an invalid value is typed into the text field, it will have no effect. Once the level has been set the controller presses send to fire off the message. The entry and scale widgets must maintain a consistent view of one underlying value. The send button will have to sample it to generate a value. We therefore have multiple views of some data.

4.1 Representing State in FranTk

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 use them within the `GUI` monad.

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

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

It is possible to get a behavior from a BVar.

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

The behavior therefore represents the flight level value at any give point in time.

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

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

The event from a `BVar` generates an occurrence every time the value of the `BVar` is updated.

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

The use of BVars is therefore a fundamental and important feature of FranTk. They provide us with a mechanism to represent state, but to use it in a functional style.

4.2 Using State in FranTk

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

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 later.)

```
scale :: BVar Int -> Component
```

```
scale bv =
 mkHScale
  [scaleValB (bvarBehavior bv)] (...)

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

As shown in section 3.2 we can tell a widget to show a string behavior using textB, and we transform the integer behavior into a string behavior using lift1. (We will fill in the rest of the definition later.)

```
entry :: BVar Int -> ... -> Component
entry bv =
 mkEntryRtrn
  [textB (lift1 show (bvarBehavior bv))]
  (...)
```

We can therefore easily provide multiple views of the state of an application.

4.3 Listeners - Updating State in FranTk

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` with 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 complete the definition of the slider as follows.

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

The function `mkHScale` takes a listener argument, which is passed the current value every time the slider updates the `BVar` with its changed value. The slider simply updates the value of the BVar with its changed value.

The introduction of listeners is a very important design choice. Initially this choice may seem strange. The use of behaviors and events encourages a more functional style of programming. However, the use of listeners introduces an imperative concept into this functional approach.

The introduction of listeners brings an important benefit. We give component-making functions a consumer (listener) argument which allows them just to yield their visual aspect in the form of a Component. This approach makes geometric composition simple. The alternative would be to return a pair of visual and semantic handles, in the form of a Widget and an Event providing access to all user input on that widget. This alternative makes component composition more complex. To compose two components we would now require to compose their Widget and Event parts. This style can become tiresome when composing complex collections of components, as programmers are forced to continually compose and break down compound events.

4.4 Composing Complex Listeners

FranTk introduces combinators that allow listeners to be composed in a functional style. As an example, consider the definition of the text entry widget in our example.

```
entry :: BVar Int -> (String -> Maybe Int)
      -> Component
entry bv parse =
 mkEntry [...]
          (comapMaybeL parse (bvarInput bv))
mkEntryRtrn :: [Conf Entry]
            -> Listener String -> Component
```

The function `mkEntryRtrn` takes a listener argument, which is passed a String representing the state of the entry widget, every time the return key is pressed.

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

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

Figure 4 shows how `comapMaybeL` works. It is a version of the standard `mapMaybe` function; however, it applies what appears to be an inverse function to a listener. This is because listeners are consumers, not producers, of values. It creates a new listener that consumes values of type `b`. When it hears a value, it applies the mapping function, and if this returns a value of type `Just a`, it passes it to its argument listener.

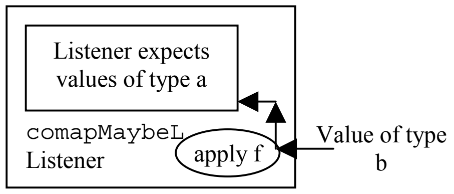

Figure 4 – The tellL listener

In the definition of entry we therefore produce a listener that parses the entry String, and only updates the application's state, if it is valid.

4.5 Sampling the State

When the "send" button is pressed, we must sample the state of the BVar and generate a message.

```
sendB :: BVar Int->Listener Int-> Component
sendB bv send =
 mkButton [text "send"]
          (snapshotL_ bv send)
```

We can make a listener snapshot a behavior and consume its current value using `snapshotL_`.

```
snapshotL_ :: BVar a -> Listener a
           -> Listener b
```

As shown in Figure 6, every time the new listener is fired it samples the behavior and passes its current value to its argument listener.

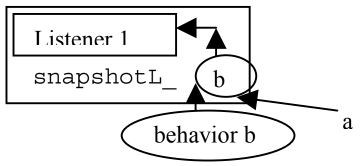

Figure 5 – The snapshot listener

This is therefore a very useful combinator, as it is often necessary to sample the state of the application when some user input occurs.

We can now complete the definition of the tactical data entry widget. It takes as parameters a callsign to display as the title of a window, an initial value to use for the level, and a listener to pass the generated message to. It creates a BVar to model the state of the proposed level, and then creates a window containing the given components.

```
levEditor :: String -> Int -> Listener Int
          -> WComponent
levEditor callsign init send = do
 bv <- mkBVar init
 mkWindow [title callsign]
   (nabove [beside (mkLabel [text "Lev"])
                   (entry bv parse),
            scale bv,
            sendB bv send])
```

```
title :: String -> Conf Window
nabove :: [Component] -> Component
```

5. THE LISTENER ALGEBRA

FranTk provides an algebra of listener combinators. Though a listener is essentially an imperative callback, this algebra allows us to treat and compose them in a functional manner. This algebra is dual to the event algebra provided in Fran. Each operation in the event algebra has a corresponding operation in the listener algebra. We will first present the most significant operations in the listener algebra, and then define formally how these relate to the event algebra.

5.1 The Listener Combinators

The null listener is `neverL`, which does nothing with any value it receives.

```
neverL :: Listener a
```

To merge two listeners we use `mergeL`, which that takes two listeners and produces a new listener that consumes values and passes them to both its argument listeners.

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

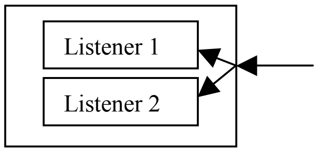

Figure 6 - The merge listener

We can therefore define, `allL`, a combinator that merges a list of listeners.

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

There is a *comap* function on listeners. In contrast to the standard map function, this applies what appears to be an inverse function to a listener. It produces a listener that consumes values, and applies the given function to these values before passing them on to the given listener.

```
comapL :: (b -> a) -> Listener a
       -> Listener b
```

A commonly used variant of `comapL` is `tellL`, which simply overrides the value a listener is passed.

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

There is a filter function on listeners. This consumes values and passes them on to the given listener, if they satisfy the given predicate.

```
filterL :: (a -> Bool) -> Listener a
        -> Listener a
```

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

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

For instance, we might require a button that could only ever be pressed once. We could define this using onceL.

```
mkOnceButton :: [ConfB Button]
             -> Listener a -> Component
mkOnceButton cs l = mkButton cs (onceL l)
```

As seen in Section 4.5, we can make a listener snapshot a behavior and consume its current value. The more general version of the snapshot function is `snapshotL`. Every time the new listener consumes a value it samples the behavior and passes the pair of values to its argument listener.

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

There is a listener equivalent of the `scanl` function.

```
scanlL :: (a -> b -> a) -> a
       -> Listener a
       -> Listener b
```

This works as shown in Figure 7. The listener's current value starts with the initial value provided. Every time the listener consumes a value *b*, it applies its update function *f* to its current value *a* and the new value, *b*. It passes (f a b) to the argument listener, and updates its current value as well.

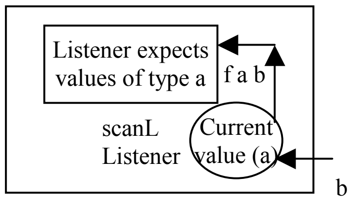

Figure 7 - scanlL

This combinator can be very useful when we require to accumulate a value every time a listener is fired. For instance, we could make a listener count up, and pass on, the number of times it had been fired using this function.

```
countL :: Listener Int -> Listener a
countL l = scanL inc 0 l
  where inc :: Int -> a -> Int
        inc n _ = n + 1
```

Previous functional toolkits would require a programmer to write this sort of code in terms of an IO action that sampled a mutable variable. In conclusion, while programmers must still use imperative callbacks in FranTk, they can manipulate them *succinctly* with a set of functional combinators, instead of being forced to write longer and more cumbersome imperative code.

5.2 Event-Listener Duality

The event and listener algebras in Fran are duals of each other. We can therefore formally define a set of relationships between them.

The primitive combination operation for events and listeners is `addListener`. This adds a listener to an event such that the listener is fired every time there is an event occurrence. This function returns a remove action that can be called to unregister the listener's interest.

```
addListener :: Event a -> Listener a
            -> IO (IO ())
```

The relationship between events and listeners can be defined in terms of the `addListener` function.

For any `e :: Event a, l :: Listener a, l1 :: Listener b, f ::a -> b,p :: a -> Bool, b :: Behavior b, l2 :: Listener (a,b), e1 :: Event b, n :: a, op :: a -> b -> a`

```
addListener neverE l <==>
addListener e neverL

neverE :: Event a

addListener (mapE f e) l1 <==>
addListener e (comapL f l1)

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

addListener (filterE p e) l <==>
addListener e (filterL p l)

filterE :: (a -> Bool)
        -> Event a -> Event a

addListener (onceE e) l <==>
addListener e (onceL l)

onceE :: Event a -> Event a

addListener (snapshotE b e) l2 <==>
addListener e (snapshotL b l2)

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

addListener (scanlE op n e1) l <==>
addListener el (scanlL op n l)

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

If we imagine a wire[1] connecting a listener to an event, the event and listener combinators can be interpreted as mechanisms to transform user input code at either end of the wire (Figure 8).

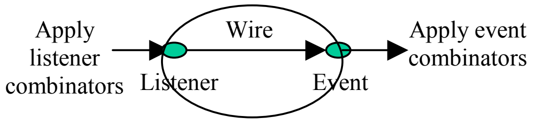

Figure 8 – A Wire

6. INTRODUCING WIRES

Sometimes we need to connect output from one component into another component. For instance, consider a button and entry widget. When enter is pressed the current text is sampled and passed to a listener.

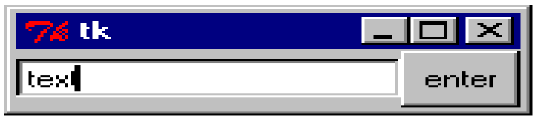

Figure 9 – Entry and Button

We can define this as follows.

```
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`. In particular, it is stateless and has no behavior. 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
```

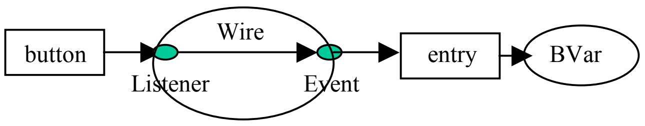

Figure 10 – Using Wires

We therefore have the following structure in our example (see Figure 10) –

- 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`.

`Wire`s and the `mkWire` functions are primitive in FranTk. We can define a `BVar` in terms of a wire.

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

newBVar :: a -> IO (BVar a)
newBVar a = do
 (l,e) <-  newWire
 let e' = a `accumE` e
 let b = a `stepper` e'
 return (BVar l e' b)
```

The definition of a BVar relies on two Fran combinators. The BVar hears update function values on the wire. These updates will modify its state.

It accumulates an event based value using `accumE`. This will therefore form an event that produces an occurrence on every update, by applying the update function to the current BVar value.

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

[1] Fran introduced the notions of listeners, events and wires in version 2. However, it did not provide any of the listener combinators discussed in this paper.

The function `accumE` is defined in terms of `scanlE`.

```
accumE x0 change =
  scanlE (flip ($)) x0 change
```

The BVar's behavior is formed by stepping through, changing on every event occurrence.

```
stepper :: a -> Event a -> Behavior a
```

We can use this approach to define other types of BVar, such as BVar collections discussed in Section 8.

7. DYNAMIC EXAMPLES – PART I

The previous examples have shown how to implement simple interfaces in FranTk. However, they have involved only a static set of widgets on screen. That is, though the appearance of individual labels has changed, the number of labels has not. The ability to handle dynamically changing collections of components in FranTk is one of its major benefits.

There are two sorts of dynamic display we could have. The first is a simple conditional display. Here we can display one of two components depending on some state. The second sort of dynamism is the introduction of new components on to a screen. We will introduce a conditional display in this section, and then a full dynamic display in Section 8.

7.1 Conditional Displays

As an example consider the tactical data entry widget from Figure 3. It will only be displayed some of the time. It will be popped up when requested. (There are a number of different aircraft views, each which can raise the data entry window). When the "send" button is pressed it will disappear.

To handle this we provide a BVar to model the state of the window. When the "send" button is pressed the window should close, so we merge the `send` listener with the BVar's listener.

```
levWindow :: BVar Bool -> String -> Int
          -> Listener Int
          -> WComponent
levWindow bv callsign init send =
  let sendAndOpen =
        mergeL send
            (tellL (bvarUpdInput a) False)
  in
  ifB (bvarBehavior bv)
      (levEditor callsign init sendAndOpen)
      emptyComponent
```

We 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` produces a component which behaves as `w1` when `b` is True and `w2` otherwise. (Other members of the `GBehavior` class include `Behaviors` and `Events`.) In this example we display the `levWindow` when the BVar has the value `True` and an empty component otherwise.

```
emptyComponent :: Component
```

This provides us with our first mechanism for dynamically altering the number of widgets on screen at any given time.

8. DYNAMIC EXAMPLES - PART II

We will now consider a dynamic example with a variable number of components on screen at any given time. The ATC system must model the state of all the aircraft in a given airspace sector. This set of aircraft will clearly change over time. We therefore need to maintain some model of a dynamic collection of aircraft. The ATC simulator uses three different views of the set of aircraft (see Figure 11). It displays a radar view with a dot and some flight data for each aircraft. It displays an electronic strip for each aircraft in the Aircraft Display Window. Finally, it displays details of the selected aircraft in the Flight Data Plan window. It is therefore very important that we have one abstract model of the data that can be viewed in several ways.

8.1 Introducing Behavioral Collections

We need to define the collection of objects that are displayed on the screen. In most previous GUI systems, we would do this by performing update actions that add and delete widgets from the screen. In FranTk, we define the appearance of an interface based on some state for all time. We therefore need to be able to define the display as a function of some abstract collection type. We do this using a behavioral collection, in this case a list.

For instance, the radar view consists of canvas components, with one `aircraftView` for each aircraft. Similarly, we have one strip in the aircraft display window for each aircraft.

```
aircrafts :: ListB Aircraft -> CComponent
aircrafts ls = pile (fmap aircraftView ls)

aircraftView :: Aircraft -> Ccomponent

aircraftDisplay :: ListB Aircraft
                -> Component
aircraftDisplay ls = nabove (fmap strip ls)

strip :: Aircraft -> Component

type ListB a
fmap :: (a -> b) -> ListB a -> ListB
pile :: ListB CComponent -> CComponent
nabove :: ListB Component -> Component
```

In FranTk we represent a dynamic list of objects as a `ListB`. We can map functions across this list. For instance, we map the `strip` function over the list to generate a strip for each aircraft. We use `nabove` to compose a dynamic collection of components above each other. We use `pile` to compose a dynamic collection of canvas components into a complex view.

In both cases, when rendered the `ListB` will incrementally update the screen only making necessary changes, rather than redisplaying everything.

8.2 Making list collections

To make a list collection we use a special type of `BVar`, a `ListBVar`. We can create a `ListB` from an initial list and an update event. It begins by behaving as the initial list, and then on every event occurrence, changes by applying the update function from the occurrence.

```
mkListB :: IList a
        -> Event (IList a -> IList a)
        -> ListB a
data IList a
```

This is therefore similar to the Fran behavior combinator, `stepAccum`, which creates a piecewise constant behavior that is updated by event occurrences.

```
stepAccum :: a ->Event (a -> a)->Behavior a
stepAccum a e = stepper a (accumE a e)
```

The `IList` type is a special incremental list type that maintains incremental updates. The Haskell Edison library [8], defines a general interface for dealing with functional data structures such as `Sequences` and `Sets`. The `IList` type implements the `Sequence` interface, allowing us to treat them in the same way as standard lists. This therefore provides a powerful, and familiar set of operators for constructing dynamic lists. We can therefore generate values of type `IList` using Sequence functions such as empty, single, cons and append.

FranTk also supports the notion of a list behavior variable, or `ListBVar`.

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

When creating a `ListBVar` we give it an initial list of values. We have standard functions to extract the *value, input listene*r, and *update-input listener* from a `ListBVar`. Here updates are also defined using the IList type.

```
collection :: ListBVar a -> ListB a
bvUpdInput :: ListBVar a
           -> Listener (IList a -> IList a)
bvInput :: ListBVar a -> Listener (IList a)
```

These behavioral collections are powerful. They allow us to declaratively handle dynamic collections of data. In the next section we demonstrate how to filter a dynamic collection.

8.3 Sorting list collections

It is often necessary to apply functions such as sort and filter across lists. FranTk provides the ability to sort a dynamic list based on some dynamic predicate. For instance, in our example the strips in the Aircraft Display window can be sorted in a range of ways, such as by flight level. We can do this using `sortByB`.

```
sortByB :: (a -> Behavior b)
        -> Behavior (b -> b -> Ordering)
        -> ListB a -> ListB a
```

This takes a function to extract a behavior from a list element, and a predicate valued behavior and applies these to filter the list. In our example, the predicate would be set by the "Sort by" menu. The extraction function is needed because the elements of a dynamic list may themselves be dynamic. The Aircraft type consists of a set of individual behaviors representing the parameters of the plane.

```
data Aircraft = Aircraft {
      flightLevB :: Behavior Int, ... }
```

From these we need to extract a single behavior representing the relevant parameters needed for sorting.

```
extract :: Aircraft -> Behavior Extract

data Extract = Extract {
    flightLev :: Int, ...}
```

This style is very important and represents a common pattern for many FranTk programs.

8.4 Summary

In FranTk we can therefore treat dynamic collections in a similar manner to static collections, modeling them as values. This allows us to easily manipulate them, and create multiple views of the same collection. For instance, in our current example we have one sorted view of the dynamic list. In FranTk, we can easily construct dynamic interfaces in a declarative programming style. This makes it particularly easy to implement multi-user interfaces; however, it is important in many other styles of application.

9. LARGE EXAMPLES

FranTk has been applied to a range of examples including the prototype Air Traffic Control simulator discussed in this paper (and shown in Figure 11), and a structured program editor.

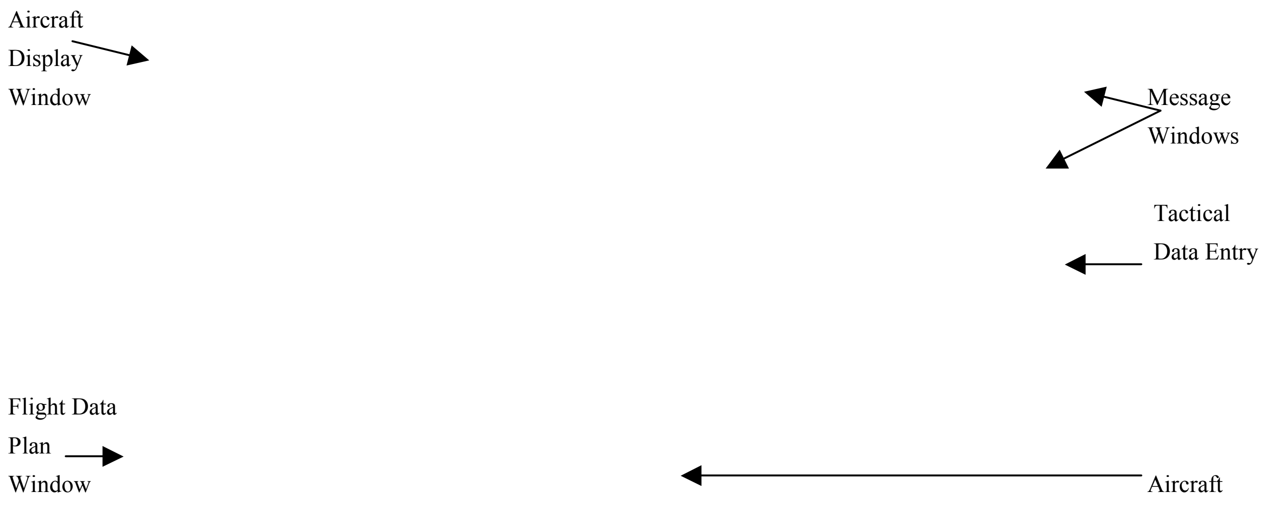

Figure 11 - The Prototype Air Traffic Control Simulator

9.1 The ATC System

In general, the development of a large, complex case study was relatively easy in FranTk. We were able to construct the system using a consistent programming approach. The system was designed in terms of a set of components, which were integrated in a compositional manner.

The simulation is described as a collection of behaviours (representing aircraft and sectors) that communicate via events. Each adjacent sector and aircraft is modelled as a function which accepts messages via an event stream, and produces an event stream generating a set of response messages. Each aircraft maintains an abstract trajectory model. The use of Fran behaviors proved very useful when developing the ATC system. Using behaviours, we were able to provide a simple, elegant model of an aircraft's trajectory. The aircraft model then simply snapshots the appropriate flight parameters when it needs to generate a flight-parameter downlink message (which tells the controller where the aircraft is). The active aircraft are then modelled as a dynamic set. New aircraft are created on the basis of an alarm event, which goes off according to a plan (read in from an input file). Aircraft are deleted according to a predicate, which specifies when the aircraft leaves the user's screen, and therefore ceases to be useful. FranTk's support for real-time predicates therefore proved particularly useful when developing the prototype. Predicates are also used, for instance, to define time outs on messages.

The use of dynamic collections to model the collection of aircraft, and datalink messages was again important. (A Datalink message is an electronic messages sent from a controller to an aircraft to inform it to change trajectory). The system provides a number of different views of an aircraft's data and of the datalink message collection. For instance, the "Message In", "Message Out" and "Datalink Msgs" windows each show a separate filtered view of the sector's datalink message set. The ability to provide multiple views of a dynamic collection was therefore very important.

The ATC system was a large case study consisting of several thousand lines of code. It runs at a usable speed, simulating several aircraft, when compiled under GHC. It was developed as a prototyping exercise in co-operation with a "human factors" specialist at the UK's National Air Traffic service. The resulting interface proved useful for him, and he requested a copy of the final system.

9.2 A Structured Editor

FranTk has been used to develop a simple declarative implementation of a structure editor for a small imperative language[2]. Bernard Sufrin and Oege De Moor have developed a simple, purely functional, model of a structured editor. This model was developed as an executable formal specification. They model the editor as a Tree, which accepts update operations.

The FranTk implementation maintains their simple, declarative model of the editor. It models the state of the editor using two parts, a Behavior Tree, modelling the current state, and a wire, upon which the Behavior is based. The wire hears about tree updates. This is therefore a special form of BVar used to represent the tree status. The status of a text editor is itself represented by a document behavior. We therefore provided a simple mapping between the Tree updates, and declarative structured document updates. We were therefore able to develop a relatively simple, very high level implementation of a structured editor. Though high level, the implementation is still efficient. It runs at a usable speed, even when only run with Hugs, the Haskell interpreter.

[2] This was joint work with Oege De Moor from Oxford University.

10. IMPLEMENTATION DETAILS

The efficient implementation of FranTk relies on three key features. It uses a data driven programming model. It uses weak references to limit the amount of work needing done. It uses an incremental implementation for behavioral collections. In this section we will briefly summarize these key features.

10.1 Data driven implementation

A simple implementation of events and behaviors requires that behaviors and events are sampled every time interval. This would be prohibitively expensive in a large user interface, as every aspect of the interface would need to be redisplayed every time any input was received. Instead FranTk, uses a data driven model. Events and behaviors have invalidation actions associated with them. When the listener talking to the event or behavior is fired the invalidation action is performed. After any user input only those components that rely on behaviors or events that have been invalidated need to be redrawn.

10.2 Finalisers & Weak References

Once a BVar has been created, the listener will begin talking to the BVar, passing on every value it hears. This is useful only so long as the event or behavior from the BVar is in use. However, often these will only be used for a fraction of the lifetime of a program. For instance, if a component were later removed from the screen and the behavior it relied upon was no longer used it would be useful to remove the listener as well, to prevent unnecessary work. For this purpose, we use *weak references* and *finalisers* [11]. Weak references enable listeners to talk to events without keeping them alive in the heap. Finalisers are actions which can be added to an object, and which will run when the object is garbage collected. This mechanism is used to delete listeners. When the clients (the events and behaviors) that a listener can talk to are all garbage collected, a *finaliser* will be run to delete the listener.

10.3 Incremental behavior collections

The implementation of efficient dynamic collections requires some extra work. A collection behavior is considered to consist of two parts, a simple behavior representing its value at any given time, and an event generating individual incremental changes.

For instance, a `ListB` consists of the following parts.

```
data ListB a = ListB (Behavior [Entry a])
                     (Event (ListUpdates a))

data ListUpdates a =
  LUpds [ListUpdate] | ResetL [Entry a]

data ListUpdate a = InsertL [Entry a] Pos
                  | DeleteL Ident
                  | MoveL Ident Pos

data Entry = Entry Ident a
data Ident
```

The `Ident` data type represents unique values. Each element in the list has a unique value associated with it. Incremental changes involve inserting an element at a given point in a list, moving an element to another position in the list, deleting an element or resetting the list. Recall that updates to a List behavior are made using functions of type `IList a -> IList a`. These updates can clearly be based on far more sophisticated notions that simple equality. The notion of unique identity is therefore very important.

The `IList` type deals with generating unique names for entries, and for generating these updates. An `IList` consists of a list of tagged entries, a name generator to make new named entries and a set of current updates. Combinators such as `cons` require to generate a name for their new element, add the new element to the list, and generate an appropriate update.

```
data IList a =
 IList [Entry a] NameGenerator
       (ListUpdates a)
type NameGenerator =
 a -> (Entry a,NameGenerator)
```

To render the list on to an interface we therefore make changes corresponding to each incremental update.

11. RELATED WORK

There have been a number of previous approaches to functional GUI toolkits.

The *TkGofer* [14] toolkit is also built on top of Tcl-Tk. As with FranTk, components are given lists of configuration information, and type classes are use to restrict which configuration options can be given to which components. We have extended this approach with dynamic configuration options. *TkGofer* provides a much lower level interface to GUI programming, relying on callbacks. This results in a more imperative style of programming with all state being stored in mutable variables.

Toolkits based around concurrency, such as *Haggis* [4], have been developed that avoid the need for callbacks. *Haggis* provided a structured declarative approach to specifying pictures. However, this only worked for static images. *Haggis* treated dynamic widgets separately in a more imperative manner. Haggis separated a widget into a typed value that could be used to update it, and an untyped handle that could be used to display it. In contrast, we have one representation of a widget, the *Component* and pass all configuration information as parameters when creating the component. By passing in a listener as a parameter to the component, we avoid the need for a handle to access user input.

The *Fudgets* toolkit [1] attempted to support a more declarative approach to building user interfaces. A Fudget is a value of type F a b that receives messages of type a and sends messages of type b. Fudgets can be composed to make more complex fudgets in a functional style. However, there is only limited support for creating interfaces with a dynamic number of components on screen. Fudgets also makes it difficult to support application and interface separation, making it difficult to have multiple views of the same data [5].

The Clock [5] and Pidgets [12] toolkits are the most similar to our own. Clock is based on the well-known MVC [7] architecture. Clock programs are structured as a tree of components. Each component has a model which represents its state, and which can accept updates (similar to our listeners), and requests (similar to our behaviors). The component has a view function which describes its appearance. This is defined as a function of the state (using requests). A component can take updates which represent user input. The tree structure represents the hierarchical decomposition of the interface. Components may have sub-components which they use when defining their view. The state in the model is visible to its component and all its children. Clock therefore allows dynamic views to be defined in terms of dynamic collections. However, the Clock architecture is overly restrictive making it difficult to construct interfaces with complex application behavior. The structure also makes it very difficult to define parameterised components with local state.

Pidgets is a toolkit based around a "monad of imperative streams". An imperative streams program is represented as a term of type `St a`. When run this will produce a value of type a at repeated points during its execution. A stream produces values. However, it also has a current value, which is the last value it produced. A stream may also perform IO actions while producing its value. An imperative stream therefore unifies our separate concepts of Behaviors, Listeners and Events. This conceptual combination can make Pidgets difficult to deal with. For instance, it requires two new operators *start* and *next* to be introduced into the language.

```
start :: St a -> St (St a)
next :: St a -> St a
```

The next operator takes a stream and returns one that ignores any values before the next time step. Starting a stream conceptually starts it off as concurrent process. The start function returns a new stream which outputs a value every time the concurrent stream produces one. This concept is important when IO actions are concerned as these will be performed by the concurrent process and not by the returned stream. These two new operators can rapidly become difficult to handle.

To give an impression of how FranTk compares to Object Oriented approaches we will compare it to Java's Swing [6]. To provide support for application/interface separation, Swing has embraced the Model-View-Controller paradigm. User Interface components can be associated with abstract models. For instance, there is a `ListModel` which defines the methods which components such as Swing listboxes use to access lists. These models support the concept of Listeners. Java's Listeners, in fact, inspired the notion of listener in Fran/FranTk. For instance, we can add a `ListDataListener` to a `ListModel` to find out about any changes to its state. This is therefore similar to use of the primitive `addListener` function in FranTk.

However, the notion of listener in FranTk is more general and more powerful. Java distinguishes listeners based on what they can be added to. This means that it is sometimes difficult to add a listener in two places. For instance, Java distinguishes between `MouseListener's` which hear mouse clicks and `ActionListener's` which hear action events. FranTk, however, would consider both to be of type `Listener ()`. We would therefore need to duplicate code in Java. In addition, in Java, to consume values of new types we must define new Listener classes. In contrast, FranTk listeners can be parameterised over any type.

Conversion between listeners is also more cumbersome in Java. For instance, if we had an item listener (which hears about selected objects), and wanted it to be fired every time a button was pressed, we would require to write the following code. It creates a new Action listener, which fires the item change listener.

```
ActionListener l = new ActionListener () {
```

```
public void actionPerformed(ActionEvent e)
{i.itemStateChanged (new ItemEvent
            (e.getSource(),
             ItemEvent.ITEM_STATE_CHANGED,
             obj, ItemEvent.SELECTED))}};
```

In contrast, in FranTk we would simply use `tellL` to convert the listener: `tellL itemListener obj`.

Java's models suffer from a similar restriction. Again they are less generic than BVars, which can be used to hold values of any type. Java's `ListModels` are also less powerful than FranTk's. To write the equivalent of FranTk's dynamic sorting function would take significantly more code.

In conclusion, Java's Swing does provide more capabilities than FranTk. It also, like all imperative approaches, provides more low-level control. This makes it easier to ensure efficiency. However, for many applications FranTk's combinators make it more succinct.

12. CONCLUSIONS

We have presented FranTk, a toolkit for developing graphical user interfaces in Haskell. It concentrates on providing a programming model that is both "declarative in the large and in the small".

The state of an application can be defined as a behavior value. These values can be easily composed. Unusually this extends to the ability to handle dynamic collections of objects as values, treating them in a functional manner.

FranTk introduces the concept of a listener as an abstract value. Listeners allow imperative actions to be handled, but composed in terms of a functional algebra.

FranTk defines an interface in terms of components. These are constructed by passing in configuration options, including dynamic options. This allows us to define a component's appearance for all time. A Listener argument is also passed in when creating a component, thereby separating the semantic wiring from the visual Component. These components can therefore be geometrically composed using simple, pure functions. However, a Component represents an action that produces a widget. This allows it to have its own internal state.

FranTk therefore allows a compositional, declarative style of programming with both static and dynamic user interfaces. It has been developed on top of Tcl/Tk providing a set of platform independent, powerful widgets. However, it has been implemented in a very toolkit independent manner, making it easier to port to other GUI toolkits. It has been released as a publicly available toolkit (http://www.haskell.org/FranTk). Finally, it has been applied to a range of large examples, including a structured text editor, and an air-traffic control simulator.

13. ACKNOWLEDGMENTS

Thanks to Conal Elliott and Simon Peyton Jones for valuable comments on this paper. Thanks also to Conal for helping me work on the Fran core implementation, and to Simon Peyton Jones and the GHC group at Microsoft Research where I was employed as an intern to complete work on FranTk. Finally, thanks go to the UK's National Air Traffic Services for co-operating with the ATC case study.

14. REFERENCES

[1] M Carlsson, T Hallgren: FUDGETS – A Graphical User Interface in a Lazy Functional Language, *Conference on Functional Programming Languages and Computer Architectures*, 1993.

[2] J Coplien, D Schmidt*: Pattern Languages of Program Design*, Addison-Wesley 1995.

[3] C Elliott, P Hudak: Functional Reactive Animation, *International Conference on Functional Programming 1997 (ICFP'97)*.

[4] S Finne, SL Peyton Jones: Composing the User Interface with HAGGIS, *Summer School on Advanced Functional Programming*, Olympia, WA, Aug 25-30, Springer Verlag LNCS, 1996.

[5] TCN Graham*, Declarative Development of Interactive Systems*. Volume 243 of Berichte der GMD. Munich: R. Oldenbourg Verlag, July 1995.

[6] Sun Microsystems, The Java Swing Connection, available at http://java.sun.com/

[7] GE Krasner and ST Pope, A cookbook for using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1 (3):26-49.

[8] BA Myers, DA Giuse, RB Dannenberg, BV Zanden, DS Kosbie, E Pervin, A Mickish, P Marchal, Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. In *IEEE Computing*, pages 71-85, November 1990.

[9] Chris Okasaki, *Edison User's Manual*, available at http://www.haskell.org/.

[10] J Ousterhout: *Tcl and the Tk Toilkit*, Addison-Wesley, 1992.

[11] S Peyton Jones, S Marlow, C Elliott: Stretching the Storage Manager: weak pointers and stable names in Haskell, in *Implementing Functional Languages 1999*.

[12] E Scholz: Imperative Streams – A Monadic Combinator Library for Synchronous Programming, *International Conference on Functional Programming 1998 (ICFP'98)*.

[13] B. A. Sufrin and O. de Moor. Modeless structure editing. In: J. Davies, A. W. Roscoe and J.C.P. Woodcock (editors), Proceedings of the Oxford-Microsoft symposium in Celebration of the work of Tony Hoare, September 13-15, 1999.

[14] T Vullinghs, D Tuijnman, W Schulte: Lightweight GUIs for functional programming, *Programming Languages: Implementations, Logics and Programs*, 7th International Symposium, Springer Verlag LNCS, 1999.