Structuring Graphical Paradigms in TkGofer

Koen Claessen
OGI and Utrecht University
koen@cse.ogi.edu

Ton Vullinghs
Universität Ulm
ton@informatik.uni-ulm.de

Erik Meijer
OGI and Utrecht University
erik@cse.ogi.edu

Abstract

In this paper we describe the implementation of several graphical programming paradigms (Model View Controller, Fudgets, and Functional Animations) using the GUI library TkGofer. This library relies on a combination of monads and multiple-parameter type classes to provide an abstract, type safe interface to Tcl/Tk. We show how choosing the right abstractions makes the given implementations surprisingly concise and easy to understand.

1 Introduction

In his article 'Why Functional Programming Matters' [7], John Hughes explains that an important feature of a programming language is the way in which a language provides *glue* for combining building blocks to form larger structures. The better the glue, the more modular programs can be made. He argues that functional languages offer very powerful kinds of glue like higher order functions and lazy evaluation. Further research evolved a new kind of glue: type and constructor classes [9].

Unfortunately, there used to be a separation between the imperative world, which had very useful bricks, but no glue, and the functional world, which had powerful glue, but no bricks. Recently however, things changed when *monads* were introduced into functional programming [13, 18]. When using monads it becomes possible to seamlessly introduce imperative features into a purely functional language.

This technique is used in *TkGofer*, a library in Gofer [8] that interacts with the imperative graphical toolkit Tk [12]. On top of a few new primitive functions, a system is built that offers the programmer structural access to the widgets of Tk. But, unlike Tk, widgets in TkGofer are typed and type classes are used to group different kinds of properties for widgets. This provides the functional programmer with a powerful tool for performing GUI programming.

A criticism of the use of monads is that they lead to an imperative style of programming. A more functional way of coding GUIs without monads is, for example, Fudgets [1]. We think, however, that monads are more general and offer more structure. This is shown by giving an elegant implementation of Fudgets and ActiveX Animation [3]. Furthermore, we show that TkGofer is capable of expressing other well known graphical paradigms, such as the Model View Controller [10].

There are also other systems that try to integrate graphical user interaction in functional languages. Examples are Fudgets [1], Haggis [5] and sml_tk [11]. We believe that they do not capture the essence of what makes functional GUI programming concise, elegant, and powerful. They lack using monads for imperative actions, combined with type classes for structure. As we shall see, constructor classes and multiple-parameter type classes play a critical role in our system.

The remainder of this paper is organized as follows. The next section of this paper introduces TkGofer. The third section describes the integration of the Model View Controller paradigm in TkGofer. The fourth section shows an implementation of Fudgets. The implementation of a functional animation toolkit is discussed in section five. The last section compares TkGofer to a few other graphical functional systems and discusses our results.

2 TkGofer

In this section we give a short introduction to TkGofer. For a more detailed description of TkGofer we refer to other papers [16, 17].

2.1 Creating a GUI

How do we write GUIs in TkGofer? To explain this we present a small example. Figure 1 illustrates a picture of a decimal counter and the code that implements it. The user interface shows five widgets: a window, a label, an integer display field, and two command buttons.

The functions `window`, `entry`, and `button` create widgets — each with a specific list of configuration options. The effect of performing these actions is that Tcl/Tk creates a handle for the new widget. Via this handle we may modify the widget or specify its layout. The layout of the GUI is constructed using layout combinators. For example, the combinator `(<<)` combines two widgets horizontally. The com-

```
counter :: GUI ()
counter =
  do w <- window [title "Counter"]
     l <- label  [text "Value:", background "green"] w
     e <- entry  [initValue 0, background "yellow"] w
     p <- button [text "-", command (updValue pred e)] w
     s <- button [text "+", command (updValue succ e)] w
     pack (l ^-^ (p << e << s))
```

Figure 1: *A decimal counter*

binator (`^-^`) combines two widgets vertically and aligns them. The function `pack` actually displays the combined widgets on the screen. When the user presses the '`+`' button, the function `updValue` is invoked. This function replaces the value of the entry field by the incremented value (see also the next section). Likewise, if the user presses the '`-`' button, the value is decremented .

The functions `button` and `entry` have the following signatures:

```
button :: [Conf Button]
       -> Window -> GUI Button
entry  :: GUIValue a
       => [Conf (Entry a)]
       -> Window -> GUI (Entry a)
```

The signatures reveal some interesting characteristics of TkGofer. First, to provide widget objects with a unique identity, they are created within the `GUI` monad. This monad is an extension of the standard `IO` monad. For the remainder of the paper it is sufficient to assume that values of type `GUI a` represent actions that may have some side effect on the user interface (e.g. close a window) and return a value of type `a` (e.g. the contents of a text-editor). Second, widgets that handle user input are typed over their contents. This type has to be an instance of the class `GUIValue` that defines parse and unparse methods. Third, unlike many other GUI libraries where information is just coded into strings, TkGofer widgets have typed configuration options. Configuration options specify the external appearance and behaviour of a widget. Although many differences between widgets exist, a lot of options are shared, e.g., the function `background` applies to both the entry and the button widget. In Sect. 2.2 we explain how we use type classes to express the common characteristics between different widgets.

Another aspect, which is not obvious from the above signatures, is that we distinguish several kinds of widgets. A widget is either a window item, a menu item, a canvas item or a toplevel item. This means that a widget may appear on a window, in a menu, in a canvas or may act as a container for other widgets. Type classes may be used to express this property. The signatures for window items are generalized by the following type synonym:

```
type Wid w = [Conf w] -> Window -> GUI w
```

The signature of `button` and `entry` may now be written as `Wid Button` and `GUIValue a => Wid (Entry a)`, respectively.

2.2 The TkGofer Widget Hierarchy

To constrain functions to a specific class of widgets we introduce a hierarchy of type classes (Fig. 2) .

The root class, `Widget`, contains methods that apply to every widget, e.g., the function `cset` to update a configuration of a widget.

```
class Widget w where
  cset :: w -> Conf w -> GUI ()
  ...
```

All other classes in the widget hierarchy are subclasses of `Widget`. The class `HasCommand`, for example, includes all widgets that may be configured with a callback function.

```
class Widget w => HasCommand w where
  command :: GUI () -> Conf w
```

As we saw in Fig. 1, buttons are the prototypical instance of this class.

```
instance HasCommand Button where ...
```

To set the command of a button `b` to the function `cmd`, we can write

```
cset b (command cmd)
```

because every instance of `HasCommand` also has the `cset` method.

Another example of a type class is `HasInput`. In this multiple-parameter type class we group widgets that can handle user input. Its methods are overloaded on both the widget type (like `HasCommand`) as well as on the input type.

```
class Widget (w v) => HasInput w v where
  getValue :: w v -> GUI v
  setValue :: w v -> v -> GUI ()
  updValue :: (v -> v) -> w v -> GUI ()
```

An example instance of this class are entry fields.

```
instance GUIValue v => HasInput Entry v
  where ...
```

The values that can be displayed in an entry field are restricted to instances of the class `GUIValue` (e.g. `String` and `Int`).

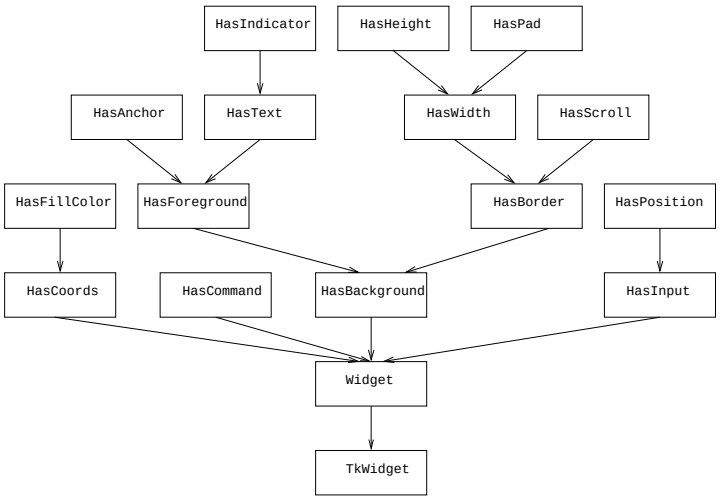


Figure 2: *The TkGofer widget hierarchy*

2.3 Extending TkGofer

Using the TkGofer primitives, it is possible to present the whole functionality offered by Tcl/Tk to the Gofer programmer. But we can go further; using higher order functions and algebraic datatypes we can easily roll our own custom widgets. Making such abstractions is hard to do in Tcl/Tk and in most other GUI Libraries.

As a concrete example of a composed widget, we build a spin button [6]. A spin button is often used to control a value that can be incremented and decremented. The spin button widget actually consists of two other widgets: a 'spin-up' button and a 'spin-down' button (see Fig. 3).

Figure 3: *The spinbutton widget*

How do we implement this new widget? First, we define a type for the new widget.[1]

```
type SpinButton = WItem (Button, Button)
```

We use the predefined type constructor `WItem`, to indicate that this is a widget that can be placed in a window. A `SpinButton` now also contains some information about its graphical properties. We can access the two buttons by using the function `witem`.

```
data WItem w = ...

witem :: WItem w -> w
```

[1] In Gofer, a type declaration will only create a new type if it is a datatype or a restricted type. To make the presentation clearer we will use type synonyms as if they worked as restricted types, so that they can be made instance of constructor classes.

The exact layout of the widget is specified in the construction function `spinButton`.

```
spinButton :: Wid SpinButton
spinButton cs w =
  do b1 <- button [bitmap "up.bmp"] w
     b2 <- button [bitmap "dn.bmp"] w
     composeWidget (b1,b2) (b1 ^^ b2) cs
```

The function `composeWidget` takes care of the extra administrative information for the `WItem` structure.

We have to make sure that the configuration options are correctly distributed over the components of the composed widget. Therefore, we overwrite the default method for `cset`.

```
instance Widget SpinButton where
  cset w c =
    do cset (fst (witem w)) (const (c w))
       cset (snd (witem w)) (const (c w))
```

Finally, we make `SpinButton` an instance of the classes `HasBackground`, and `HasForeground` and we define the widget specific configuration options.

```
spinUp, spinDn :: GUI () -> Conf SpinButton
spinUp c w = option
  (cset (fst (witem w)) (command c))
spinDn c w = option
  (cset (snd (witem w)) (command c))
```

In exactly the same way, the combination of a spin button, an entry field, and a label can be hidden in a new widget. We will call this widget the `Spinner` widget.

```
type Spinner v =
        WItem (Entry v, Label, SpinButton)

spinner :: PredSucc v => Wid (Spinner v)
spinner cs w =
  do e <- entry [] w
     l <- label [] w
```

```
s <- spinButton
       [ spinUp (updValue succ e)
       , spinDn (updValue pred e)
       ] w
composeWidget (e,l,s)
  ((e ^-^ l) << s ) cs
```

The label is used to display some text string. The spinner widget only controls values which are an instance of the class `PredSucc`. This class defines the functions `pred` and `succ` to calculate predecessor and successor values for some type `a`.

We make `Spinner` an instance of the classes `Widget`, `HasText`, `HasCommand`, and `HasInput` and we overwrite the methods `getValue` and `setValue`.

```
instance PredSucc v => HasInput Spinner v where
  getValue = getValue . fst3 . witem
  setValue = setValue . fst3 . witem
```

If we use the spinner widget instead of the two command buttons, we can reimplement the example of Fig. 1. The new implementation and a picture (see Fig. 4) are given below.

```
counter2 :: GUI ()
counter2 =
  do w <- window []
     s <- spinner [text "value", initValue 0] w
     pack s
```

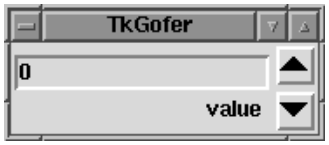


Figure 4: *A spinner: a counter with a spin button*

Composing widgets is a flexible way to construct reusable building blocks. Composed widgets can be integrated in the widget hierarchy, thus inheriting the properties of their components.

3 The MVC Paradigm

The basic widget layer of TkGofer, and the ability to extend this layer, allows us to write further abstractions. One of these is the Model View Controller paradigm (MVC). The MVC paradigm is one of the oldest object-oriented design paradigms [10]. It provides a modular way to represent information (Model), to display information (View) and to control the interactions with the information (Control). The main idea is to support several views of the same data. Changes in one view should be reflected in all the other views. As an example, consider a debugger that offers functions to display a syntax tree. The tree represents the model. Possible views are a textual and a graphical representation of the tree. The controller keeps the two views consistent (see Fig. 5).

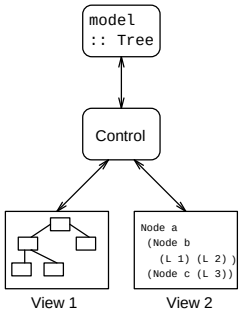


Figure 5: *The MVC paradigm*

3.1 The Controller

A controller has to know which model (value) it has to observe. If a certain view changes, the view will send the controller a message containing the new value. Subsequently, the controller will send a message to all dependent views to tell them they have to change as well. Thus, the controller also has to know which views it has to control.

We represent the ability to send a message by a function with the following type:

```
type Send a = a -> GUI ()
```

When we apply this function to a message of type `a`, it returns an action of type `GUI ()`, which will, when executed, send the message. This concept is sometimes called a *self addressed envelope*. The controller gets an envelope (of type `Send a`) from the view, that is used to return messages (of type `a`) to the view.

We define the controller as an abstract widget, i.e. a widget without a graphical representation.

```
type Control v = (State v, State [Send v])
```

The first component of `Control` is an updatable state containing the value of the model. The second state keeps a list of dependent update methods for the model. States are also abstract widgets. A new state is created using the function `state`. Using `getValue` and `setValue` we can access the state.

The function `control` creates a new controller object.

```
control :: [Conf(Control v)] -> GUI(Control v)
control cs =
  do v <- state []
     m <- state [ initValue [] ]
     composeAbsWidget (v,m) cs
```

The function `addMethod` adds a new update method to the method list of the controller:

```
addMethod :: Send v -> Control v -> GUI ()
addMethod send (v, m) = updValue (send :) m
```

We make `Control` instance of the classes `Widget` and `HasInput`:

```
instance Widget (Control v) where ...
instance HasInput Control v where
```

```
getValue = getValue . fst
setValue (v, m) x =
  do setValue v x
     fs <- getValue m
     seqs [ f x | f <- fs ]
```

In this declaration, `getValue` will read the actual value of the model. The function `setValue` updates the model and the dependent views. The standard function `seqs :: [GUI ()] -> GUI ()` executes a list of actions in that order.

3.2 Adding Views

Different views are represented by different widgets. For each view we have to implement two functions. The function `updateView` specifies the way the controller has to send a message to update the view. The function `invokeView` is to tell the view how to send the controller a message when someone changes the view.

The functions are methods in the class `View`. This class takes two parameters. The first one corresponds to the view and the second one to the model we want to control.

```
class View w v where
  updateView :: w -> Send v
  invokeView :: w -> Send v -> GUI ()
```

Based on these two functions we introduce a new configuration option. Configuring a widget `w` with the option `mvc c` means that `w` will be controlled by controller `c`.

```
mvc :: View w v => Control v -> Conf w
mvc c w = option
  (do v <- getValue c
      updateView w v
      invokeView w (setValue c)
      addMethod (updateView w) c)
```

If a widget `w` is created and controlled by controller `c`, it will first read the value of `c` and set its own value to this value. Furthermore, `w` will install its own invoke method, so that `w` will send its displayed value to `c` as soon as it is changed. Finally, the update method for `w` is added to the method list of `c`.

As an example, we make `Spinner` an instance of the class `View`:

```
instance View (Spinner v) v where
  updateView     = setValue
  invokeView w f = cset w
    (command (do v <- getValue w
                 f v ))
```

When the user presses one of the spin controls, the spinner will read its value and send it to the controller. The spinner updates its value by performing a `setValue` with the new controller value.

3.3 Example

In the following example we use the MVC paradigm to control an integer value by a spinner widget and a scale widget. A scale widget is a widget that displays an integer value and allows users to edit this value by dragging a slider. The `View` instance for `Scale` is written in a similar way as the instance for `Spinner`. Both widgets are on the same window. On every window there is a 'copy' button. When the user presses this button, a copy of the window is made and two new views are created.

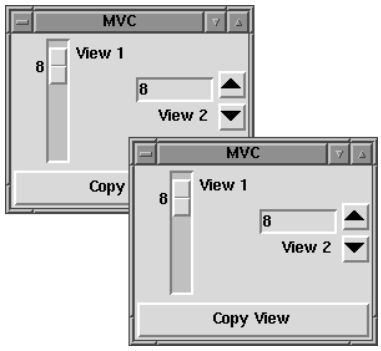


Figure 6: *The MVC example*

```
views :: GUI ()
views =
  do c <- control [ initValue 8 ]
     copy c
 where
  copy c =
    do w <- window  [ title "MVC" ]
       v <- vscale  [ text "View 1", mvc c ] w
       i <- spinner [ text "View 2", mvc c ] w
       b <- button  [ text "Copy View"
                    , command (copy c)
                    ] w
       pack ((v << i) ^-^ b)
```

The function `views` creates an initial controller, and calls the function `copy` to open the first window. Each time the button is pressed, this function is called again, thus creating a copy of the window and adding the scaler and the spinner widget to the controller.

This example only shows the basic principles of the MVC paradigm. Its benefits are of course better expressed in larger applications. The MVC paradigm tries to abstract from low level event programming and hides the communication between widgets in a few primitive functions. In the next section we will see Fudgets, which are another approach to make the communication between widgets more implicit.

4 Implementing Fudgets

In this section we will explain what *Fudgets* are [1], and discuss how to implement them in TkGofer.

4.1 The GUI Framework

The main abstraction used in the Fudgets system is called a *Fudget* (*Fu*nctional Wi*dget*). A Fudget is an object consist-

ing of two parts. One is the graphical representation of a Fudget, the other is its functionality. For example, a button Fudget is, for the user, a visible object in a window, that can be clicked. The programmer can specify the properties of such a button, i.e. what to do when the button is clicked.

A Fudget can send and receive messages of particular types: the button Fudget sends out a `Click` whenever the user presses it and the programmer can send a `Bool` to the button to change its activity status. A Fudget that receives messages of type `a` and sends messages of type `b` has the type `F a b`. Only the types of the messages determine the type of a Fudget. A button Fudget has the type `F Bool Click`, but in principle any other Fudget can have that same type.

The Fudgets system offers the programmer atomic Fudgets, such as buttons and entry fields. They can be combined, using combinators, to build larger Fudgets. Different combinators provide different ways of plumbing message streams. We will discuss two combinators: serial and cross composition.

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

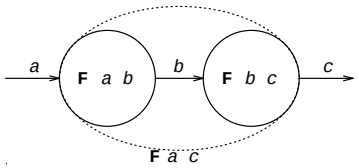


Figure 7: Serial composition

Serial composition (Fig. 7) just plugs the output stream of its right argument into the input stream of its left argument. It has a similar type as functional composition.

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

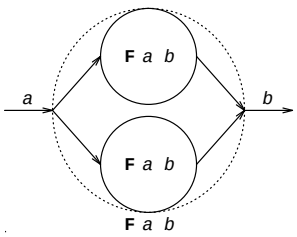


Figure 8: Cross composition

Cross composition (Fig. 8) puts its two parameters in parallell, thus forming a new Fudget. Input messages are sent to both Fudgets, the output messages of the internal Fudgets are combined to a single stream.

4.2 Fudgets in TkGofer

We will implement some of the atomic Fudgets and combinators in TkGofer. The first question is how to represent the type `F a b`.

The main task of a Fudget is to send and receive messages. We implement a Fudget as a function. The user of the function (another Fudget) tells it how to send out a message. The Fudget then returns a function that describes how to send in a message. (This part is also called the *receive* function of the Fudget). We will reuse the type `Send`, as described in Sect. 3.1.

Since we also also want to do graphical IO, we wrap this function in a monad. We call this monad the *Fudget Monad* (`FM`) and will define it later. For now, just consider `FM` as a bare `GUI` monad.

```
type F a b = Send b -> FM (Send a)
```

The next step is to define the combinators. Serial composition passes the function for sending messages to the left Fudget, which results in a function for receiving messages. This function is passed to the right Fudget as the function for sending messages, which results in the receive function for the resulting Fudget. (Actually, this is just the flipped monadic composition `@@`).

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

fudL >==< fudR = \sendC ->
  do sendB <- fudL sendC
     sendA <- fudR sendB
     result sendA
```

Cross composition is defined in a similar way. The sending function is passed to both Fudgets. The final receive function is the combination of the receive functions of the two combined Fudgets.

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

fudL >*< fudR = \sendB ->
  do lsendA <- fudL sendB
     rsendA <- fudR sendB
     result (\a -> do {lsendA a; rsendA a})
```

Notice that we do not worry about graphical IO yet, the `FM` monad will take care of that. The only thing we are concerned with here is the functionality of the Fudgets.

4.3 Atomic Fudgets

To define atomic Fudgets, such as buttons, we slightly expand the structure of the monadic type `FM`. A button needs to know in which window it is supposed to appear. Therefore, we let `FM` be a *reader monad* in the type `Window`.

```
type FM a = Window -> GUI a

instance Monad FM where ...
```

We make `FM` an instance of the monad type class. This is done in the standard way [18].

The type of a button Fudget is `F Bool Click`; it sends `Click`s and receives `Bool`s. The send function (the parameter of the Fudget) is set as a command of the button. The receive function (the result of the Fudget) modifies the activity status of the button. Other atomic Fudgets are defined in a similar way.

```
buttonF :: String -> F Bool Click
buttonF s = \send win ->
  do b <- button [ text s
                 , command (send Click)
                 ] win
     pack b
     result (cset b . active)
```

Since we added the extra `Window` parameter, we need a function that puts a Fudget in a window. For this, the Fudgets library provides the function `shellF`. We implement this function by creating a window, and passing it as a parameter to the given Fudget.

```
shellF :: String -> F a b -> F a b
shellF s fud = \send win ->
  do win' <- window [ title s ]
     fud send win'
```

4.4 Placers

Since we `pack` a widget as soon as we create it, an atomic Fudget is just put in an arbitrary place in its window. Of course, we want to manipulate the layout of the graphical objects in a Fudget. The Fudgets system has different ways of doing this. One of them is to provide layout modifiers, called *placers*, such as `horizontalP` and `matrixP`. The function `placerF` takes such a placer and a Fudget, and rearranges the layout of the internal Fudgets within that Fudget, according to the placer.

To implement this, we need to know which atomic widgets are contained in a Fudget. Therefore the `FM` type is extended to be a *writer monad* of a list of `Frames`. In TkGofer, a `Frame` is a widget with only graphical properties. Since they form the 'superclass' of all window items, every widget can be transformed into such a `Frame`. Hence, our final type for `FM` becomes:

```
type FM a = Window -> GUI (a, [Frame])
```

It is again standard to define the monadic functions for this type.

Instead of packing a Fudget as soon as we create it, we need to add it to the list of frames. When creating an atomic Fudget, we do not pack the widget, but, after some necessary lifting, use the following function instead:

```
writeFrame :: Widget w => w -> FM ()
writeFrame wid = \win ->
  do fr <- frame [] wid
     result ((), [fr])
```

A placer is basically a list-of-frames transformer. The function `placerF` applies this transformer to the internal list of frames, thus obtaining a new Fudget.

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

placerF :: Placer -> F a b -> F a b
placerF placer fud = \sendB win ->
  do (sendA, frames) <- fud sendB win
     result (sendA, placer frames)
```

There are already placer-like functions defined in the TkGofer prelude. We reuse them to obtain the following placers.

```
horizontalP frs = [horizontal frs]
verticalP frs   = [vertical frs]
matrixP n frs   = [matrix n frs]
```

4.5 Stream Processors

Plugging the message streams of one Fudget directly into another is often not what we want. Sometimes, messages are not compatible, or some intermediate computation is needed. Therefore, the Fudgets system introduces *stream processors*. SP's are just like Fudgets, but they only have a functional behavior on messages, and do not have a graphical representation. For this reason, they are also called *abstract Fudgets*. SP's are solely used to act on a message stream.

SP's are also typed by their messages. An SP of type `SP a b` receives messages of type `a` and sends messages of type `b`. SP's are defined in a continuation passing style. There are three primitives for constructing SP's.

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

The function `putSP` sends the messages in its first argument and then continues with its second argument. The SP `getSP` waits for a message, and then applies that message to its first argument, which defines how to continue. Finally, `nullSP` terminates the reception and sending of messages.

How do we implement stream processors in TkGofer? First of all, we cannot use the same type as Fudgets have. Because of the way `getSP` defines the reception of messages, an SP can redefine its behavior for incoming messages. Therefore we do not want the outside world to act the same every time a message is sent to an SP (like we did with the type `F`). An SP can change its own behavior.

Another observation is that SP's can be described as a purely functional structure; we do not need any of the extras the GUI monad offers. Therefore, we represent SP's by the following recursive datatype.

```
data SP a b = SP [b] (a -> SP a b)
```

An SP can send out some messages but then has to wait for an incoming message. The definitions of `putSP`, `getSP` and `nullSP` are now straightforward.

```
putSP mesgs (SP mesgs' fsp)
  = SP (mesgs ++ mesgs') fsp

getSP fsp = SP [] fsp

nullSP = getSP (\_ -> nullSP)
```

To use this structure, we need to turn an SP into a Fudget. The Fudgets library has a function `absF` to do this.

Because SP's can be used to create Fudgets that act like if they have state, our implementation of `absF` creates a state for each SP in TkGofer as well. In that state it keeps the SP

that is waiting for a message (having the type `a -> SP a b`). Sending a message to an SP means: apply the waiting SP to the incoming message, obtaining some outgoing messages to be sent and a new waiting SP. We put the new waiting SP back in the state, and send off all the messages.

4.6 Looping

Sometimes we want the output messages a Fudget produces to be fed back into the same Fudget. Using the combinators defined so far, we cannot do this. Therefore, some explicit combinators are introduced tying this cyclic knot. For example, the function `loopF` (Fig. 9) copies all its output messages and feeds them back as input messages.

```
loopF ::  F a a -> F a a
```

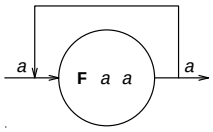


Figure 9: Looping

To define this function, we need a fixpoint operator on the `FM` monad. We use the lifted version of the generalized fixpoint operator for the `GUI` monad.

```
fixGUI :: (GUI a -> GUI a) -> GUI a
fixGUI f =
  do s <- state []
     a <- f (getValue s)
     setValue s a
     result a

fixFM :: (GUI a -> FM a) -> FM a
```

This function first makes an empty `State`, from which the result of the function `f` is going to be read. After the result is computed, it is put in the `State`. We have to be careful not to read the result before it is produced.

The reason why the traditional fixpoint on monads `fixM`, with type `(a -> M a) -> M a`, does not work here is that its parameter wants the computed value `a`, rather then the computation of that value `M a`. In this case, this is too strict, and will result in unexpected behavior.

```
loopF fud = \send ->
  fixFM (\mself -> fud (\a ->
    do self <- mself
       self a
       send a ))
```

To connect the output of `fud` with its own input, we have to change its `send` function. Now, whenever sending a message out, it should also send a message to itself. This is done by the function `self`, which is the result of the computation `mself`, provided by the fixpoint combinator.

4.7 An Example

We will use the implemented combinators to build the spinner example from Sect. 2.3.

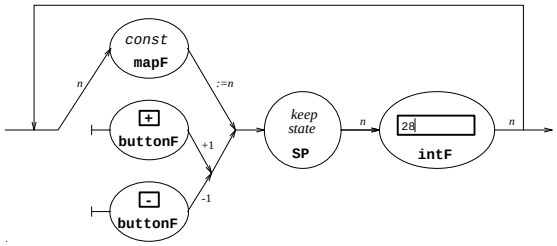
First, we define the Fudget that displays the two spin buttons. Ignoring every input message, it can send out the functions `succ` and `pred`, that is, functions of type `Int -> Int`. The buttons are placed on top of each other.

```
spinbutF :: F a (Int -> Int)
spinbutF = placerF verticalP
         ( (\Click -> succ) >=< buttonF "+"
       >*< (\Click -> pred) >=< buttonF "-"
         )
       >=< nullF  -- ignore input
```

Further, we will need a processing core, which holds the state, and applies the functions sent by the buttons. This is a stream processor, that takes an extra argument, which is the initial value of the state.

```
coreSP :: Int -> SP (Int -> Int) Int
coreSP n = getSP (\f -> let n' = f n
         in putSP [n'] (coreSP n'))
```

The main Fudget links an integer entry Fudget, the processor, and the together. Further, we add a loop to provide a connection between the output of the entry and the core — we want the user to be able to update the state as well.




```
spinF :: F Int Int
spinF = placerF horizontalP
      ( loopF ( intF
            >=< coreSP 0
            >=< (mapF const >*< spinbutF)
              ))
```

An extra feature we implemented (that is not in the original Fudgets library) is the combinator `(>=<)`. Using Gofer's type classes, it generalizes the three notions of Fudget, SP, and function in a Fudget composition. As a result we do not explicitly have to turn a function or an SP into a Fudget.

It is interesting to look at the differences between defining a new Fudget and a new widget. First, all Fudgets are instances of the type `F a b`, where a new widget, because of type safety, is always a new type. Second, the state for the spinbutton is in Fudgets stored in a separate stream processor, where in TkGofer the widget knows its own state. Third, because every Fudget has exactly one input and output stream, we sometimes end up explicitly blocking or combining streams.

4.8 Executing Fudgets

The last step is a function that executes a Fudget. We call this function `fudgeGUI`, since it translates a Fudget into the `GUI` type.

```
fudgeGUI :: F a b -> GUI ()
fudgeGUI fud =
  do (_, frs) <- fud noSend noWindow
     seqs [ pack f | f <- frs ]
 where
  noSend   = \_ -> result ()
  noWindow = error "window?"
```

It gives the Fudget a dummy window (since it is supposed to be wrapped in a window by `shellF`) and a dummy send function, and packs all the frames.

4.9 Remarks

In this section we gave an implementation of Fudgets. We believe that our implementation is very concise, and can, for example, easily be used to experiment with adding non-standard features to Fudgets. Our full implementation [2] implements many other of the basic Fudget functions and runs all the examples from the Fudgets tutorial unchanged.

During the writing of this paper we became aware of an unpublished technical report [15], in which an approach similar to ours was used to implement Fudgets [14]. However, they had difficulties with defining SP's and loops.

5 Functional Animations

In this section we will discuss a way of expressing and executing animations. This will be done in the context of a functional animation system that is heavily inspired by Conal Elliot's *Reactive Behavior Modelling in Haskell* [4]. RBMH is an implementation in Hugs of *ActiveX Animation* [3].

5.1 Behaviors

The system's main abstraction is the notion of *behavior*. A value of type `Beh a` represents a dynamic value of type `a` that can change during the animation. For example, if `Image` is the abstract type of static images, than a value of type `Beh Image` is an animation. But all other types can also be used in combination with the behavior type.

We can view a value of type `Beh a` as a *dependent* value. On what is that value dependent? Certainly on *time*, since an animation changes in time. A behavior also has a *reactive* component. This means that an action of the user, such as clicking or moving the mouse, can influence a behavior too. Therefore, a behavior is also dependent on *events*.

We represent time and events by the following types. We will not discuss here how we can use events to change behaviors.

```
type Time  = Float

data Event = LeftM
           | RightM
           | MoveM (Int,Int)
```

Another observation we can make is that time flows forward. We will never jump back to earlier points in time. Since a behavior sometimes restructures itself according to time or certain events, we want to be able to throw away old information and create a fresh behavior.

That is why we adapt the following definition for the type `Beh`. Since it is recursive, we use a datatype.

```
data Beh a = Beh (Time -> [Event] -> (a, Beh a))
```

A behavior of `a` is a function that takes a time and some events, and will return a value of type `a` plus a new behavior, to be used next time.

One of the basic functions over this type is the funtion `time`, which is a behavior over `Time`. We can use time inside an animation, for example when we want to animate a clock.

```
time :: Beh Time
time = Beh (\t _ -> (t, time))
```

Further, there exist several lifting functions. These transform functions that act on normal values into functions that act on behaviors. We gave the definition of one lifting function. The others are defined in a similar way.

```
lift :: a -> Beh a
lift a = Beh (\_ _ -> (a, lift a))

lift1 :: (a -> b)
      -> (Beh a -> Beh b)

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

The property of RBMH that we will focus on here is that animations can be specified in a functional way. Using atomic building blocks, animation modifiers and combinators we can build very complex animations in a charming, intuitive and concise way.

5.2 Images

We will use the type `Image` for static images. Images are drawn on a *canvas*, a TkGofer drawing area that can be used to display graphical objects. Another parameter images take is their place on the canvas, expressed as a coordinate. Having taken these parameters, an image will return a `GUI` action that will draw the image.

```
type Image = (Int,Int) -> Canvas -> GUI ()
```

Of course, we can add an arbitrary number of extra parameters, such as color, size, etc. To keep things simple, we will not do that here.

Using the TkGofer functions `ctext` and `cbitmap`, which draw text and bitmaps on a canvas, we can define some primitive image behaviors.

```
bitmapIm :: String -> Beh Image
bitmapIm file = lift (`cbitmap` [bitmap file])

textIm   :: String -> Beh Image
textIm string = lift (`ctext` [text string])
```

Because these behaviors are just lifted static images, they are constant animations. Combinators that act on behaviors can change this. One of them is `move`, which moves

an image according to a behavior that specifies its coordinate. Another combinator for images is `over`, which takes two images and combines them into a single one.

```
move :: Beh (Int,Int) -> Beh Image -> Beh Image
move = lift2 (\dxy img ->
         \xy can -> img (xy+dxy) can)

over :: Beh Image -> Beh Image -> Beh Image
over = lift2 (\img1 img2 ->
         \xy can -> do {img1 xy can; img2 xy can})
```

The implementation of these combinators is done with lifting.

The last function we discuss for images is the one that actually draws a (static) image on a given canvas.

```
drawIm :: Canvas -> Image -> GUI ()
drawIm can img = img (maxX/2,maxY/2) can
```

It provides the images with its parameters, using a default coordinate, which is the middle of the canvas.

5.3 Time

The structure we chose for behaviors admits another interesting feature. Because a behavior gets the time as a parameter, rather than some global variable, we can introduce local times. We can give a behavior a different time than the actual time. This principle is called *time transformation*. We present an example.

The function `later` puts a behavior *further* in time. This means that its time changes, and its events shift accordingly. All events for the behavior are delayed, in order to let them arrive at the right time. To do this, we have a function `splitStore` that takes a time and a *store* containing timed events. The function splits the store up in events that should occur now and events that will occur later.

```
type Store = [(Time, [Event])]

splitStore :: Time -> Store -> ([Event],Store)
splitStore t ((t',es):st) | t' >= t = (es, st)
splitStore t          st            = ([], st)
```

The function is a simple function that we will only use for the function `later`. It considers the store to be a queue of delayed messages. Now, we can define the function `later`.

```
later :: Time -> Beh a -> Beh a
later dt = delay []
 where
  delay store (Beh beh)
    = Beh (\t evs ->
       let t'           = t + dt
           store'       = store ++ [(t,evs)]
           (now, later) = splitStore t' store'
           (a, beh')    = beh t' now

        in (a, delay later beh'))
```

The local function `delay` keeps the store of the delayed events. It computes the new time and extracts the right events from the store. Then it gives the time and the according events to the behavior that is being delayed.

5.4 Animation

Though we defined a function `drawIm` for drawing static images on a canvas, we still do not have a function that will execute dynamic images, that is animations. Recall that animations have the type `Beh Image`.

To define the function `drawAnim`, let us first look at animations not dependent of user actions. In this case we can completely ignore the events part of behaviors.

```
drawAnim' :: Canvas -> Beh Image -> GUI ()
drawAnim' can beh = loop 0.0 beh
 where
  loop t (Beh beh) =
    do let (img, beh') = beh t
       drawIm can img
       loop (succ t) beh'
```

We create a simple loop that starts with time 0. It extracts the image from the behavior, then it draws the image, and calls itself with the new behavior for the next point in time.

An important remark should be made here. Since behaviors are infinite, this loop goes on forever. We have to be careful that the program is still able to react on user actions. The solution we chose for this, is to make a concurrent version of TkGofer. We changed the definition of the GUI monad a little bit, so that all callback actions in the system are automatically interleaved with existing actions.

So, during this loop, the system will still notice user actions, and will concurrently execute the resulting callback functions. Since adding concurrency adds a lot of additional, but standard problems, this version of TkGofer is still in an experimental phase.

Thus, to allow user interaction in the animation, we can still use a loop. Additionally, we have to collect events that happened during the animation. Mouse clicks and movements are stored in a state. We now not only pass the time as a parameter to a behavior, but we get the right events from out of the state as well.

```
drawAnim :: Canvas -> Beh Image -> GUI ()
drawAnim can beh =
  do stE <- state [ initValue [] ]
     csets can
       [ on (click 1) (updValue (LeftM  :) stE)
       , on (click 2) (updValue (RightM :) stE)
       , onxy mouse (\xy -> updValue
                              (MoveM xy :) stE)
       ]
     loop stE 0.0 beh
  ...
```

The function `drawAnim` first creates a state, in which the events are going to be collected. Then it sets some properties of the canvas, that will, in response to user actions, put the right events in the state.

```
  ...
  where
   loop stE t (Beh beh) =
     do evs <- getValue stE
        setValue stE []
        let (img, beh') = beh t evs
        drawIm can img
        loop stE (succ t) beh'
```

The loop function is a little bit extended. It now also reads and empties the event state every time a new image is drawn.

5.5 Example

We present a small example that makes use of the functions we defined. The example is made for RBMH by Sigbjorn Finne, and is called 'Time flows like a river'. It shows these five words following the mouse pointer, every word placed in time a little bit later. In this way, the words form a string that shows the history of the mousepointer (Fig. 10).

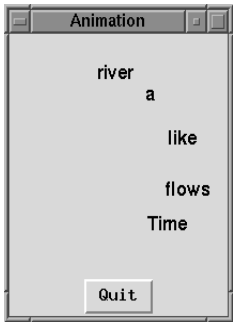


Figure 10: *Time flows like a river*

```
river :: Beh Image
river = compose (delay (follow imgs))
 where
  compose = foldr1 over
  delay   = zipWith later [0.0, 2.0 ..]
  follow  = map (move mouseXY)
  imgs    = [ textIm s | s <- words
                "Time flows like a river" ]
```

A list of images following the mouse is created. Subsequently, every element in the list is delayed. The last step is to overlay all these images.

6 Discussion

In the last two sections, we have seen ways of defining graphical structures, using atomic building blocks, modifiers and combinators. The graphical and reactive parts all relied on an imperative monad, the `GUI` monad.

Imperative monads are often accused of mimicking imperative code. But we showed that, by choosing the right types and combinators, we can neatly hide this, and exploit the expressiveness of functional programming.

And this shows the real power of monads. We can treat them as first class values, allowing us to postpone, modify and combine computations. We can even extend the monadic structure, giving us the same power as adding extra features directly to the language.

Thus, monads appear to be the ideal and most general functional object to capture imperative actions. An intuition suggested by John Launchbury is that monads give us, in contrast to normal combinators, the ability not only to compose objects, but to name the intermediate result as well. This is what makes monads so general and powerful. Over the last few years, the integration of graphical I/O and functional languages has become a popular research topic.

6.1 Related Work

Haggis [5], like TkGofer, also uses monads to handle GUIs. One of its properties is the use of concurrent processes to manage callbacks, e.g., instead of specifying what a button must do when clicked, a new process is created that waits until the user clicks the button. A confusing drawback is however that Haggis uses two different handlers for every widget; one for the graphical layout and one for the functionality of the widget.

As we saw earlier, the *Fudgets* system takes a completely different approach. Though this seems very natural and refreshing at first sight, it appears to be rather awkward after some time. Everything we want to send to a widget has to be coded into one datatype, since the Fudget type is restricted to receive messages of only one type. Furthermore, the use of the combinators restricts us to the creation of planar graphs. Often the structure of information pasing in a GUI is much more complicated than that. We end up with ad hoc methods for solving these kinds of problems.

A system that also uses Tk as its GUI basis is *sml_tk* [11]. Though provided with a nice toplevel partition in the concepts *application*, *graphical object* and *window*, the system lacks in certain points the elegance of TkGofer. Implementation details, like the creation of unique identifiers for widgets, are not hidden for the programmer, and the system is not type safe.

6.2 Conclusions

Compared to the other systems, TkGofer does have some disadvantages too. Since Gofer is interpreted, and the implementation uses character strings to communicate between Gofer and Tk, programs tend to run slower. However, in most interactive applications, this is not noticable. More often then not, the user is slower than the system. Further, many TkGofer programmers see it as big advantage to be able to use an interpreted system.

Furthermore, TkGofer does not support real concurrency. We experimented with changing the definition of the `GUI` monad to deal with coarse concurrency, at the level of primitive `IO` actions. We also added forking and communication. Though we experienced some of the standard concurrency problems, in most application this works fine, as we could see in Sect. 5.

However, the systems mentioned here lack one great feature that makes TkGofer fairly unique: *structure*. There is no hierarchy of widgets in any of these systems. For all the widgets we can specify the same properties. Either they are encoded in monomorphic functions or data constructors (Fudgets and sml_tk), or in strings (Haggis). We think this approach is unacceptable. Strong typing should be used to catch errors we make at compile time, instead of generating a run time error or unexpected behaviour.

The structure in TkGofer is primarily due to the heavy use of multiple-parameter type classes. This resulted in an extendable hierarchy of widgets, based on the functionality of the widgets. Similar structured GUI libraries where previously only available for object oriented languages (e.g., the Java AWT package). However, widget hierarchies for object oriented languages are in essence based on widget types, and therefore less intricate. Multiple-parameter classes enabled us to relate datatypes and components of datatypes. This gave us the extra glue we needed to build this tool.

The general architecture of TkGofer is not tightly coupled to Tcl/Tk, but can be reused to provide a type safe interface on top of any weakly typed library such as ActiveX or other shell scripting languages.

Acknowledgements

We wish to thank Byron Cook, Sigbjorn Finne, Jay Hollingsworth, Daan Leijen, and Wolfram Schulte for their helpful suggestions. We also thank Thomas Nordin for joining us in implementing the animation library.

References

[1] M. Carlsson and Th. Hallgren. Fudgets - a graphical user interface in a lazy functional language. In *Conference on Functional Programming and Computer Architecture*. ACM Press, 1993.

[2] K. Claessen. Fudgets Implementation. URL http://www.cse.ogi.edu/~kcclaess/Fudgets, 1996.

[3] C. Elliot. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, July 1996.

[4] C. Elliot and P. Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*. ACM Press, June 1997.

[5] S. Finne and S. Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*. Springer-Verlag, September 1995.

[6] M. Halverton. *Microsoft Visual Basic 5 Step by Step*. Microsoft Press, 1997.

[7] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 2(2), 1989.

[8] M.P. Jones. *An introduction to Gofer*, 1993. Included as part of the standard Gofer distribution.

[9] M.P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[10] G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[11] C. Lüth, S. Westmeier, and B. Wolff. sml_tk – Functional Programming for Graphical User Interfaces. Technical Report 8/96, Universität Bremen, 1996.

[12] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, 1994.

[13] S.L. Peyton Jones and Ph. Wadler. Imperative Functional Programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993.

[14] A. Reid and S. Singh. Implementing Fudgets with standard Widget Sets. In *Glasgow Functional Programming Workshop*, pages 222–235, 1993.

[15] C.J. Taylor. Embracing Windows. Technical Report TR-96-1, University of Nottingham, October 1996.

[16] T. Vullinghs, W. Schulte, and T. Schwinn. An Introduction to TkGofer. Technical Report 96-03, University of Ulm, June 1996.

[17] T. Vullinghs, W. Schulte, and T. Schwinn. The Design of a Functional GUI Library Using Constructor Classes. In D. Bjorner, M. Broy, and I. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, Novosibirsk, December 1996. Springer-Verlag.

[18] Ph. Wadler. The essence of functional programming. In *ACM Principles of Programming Languages*, 1992.