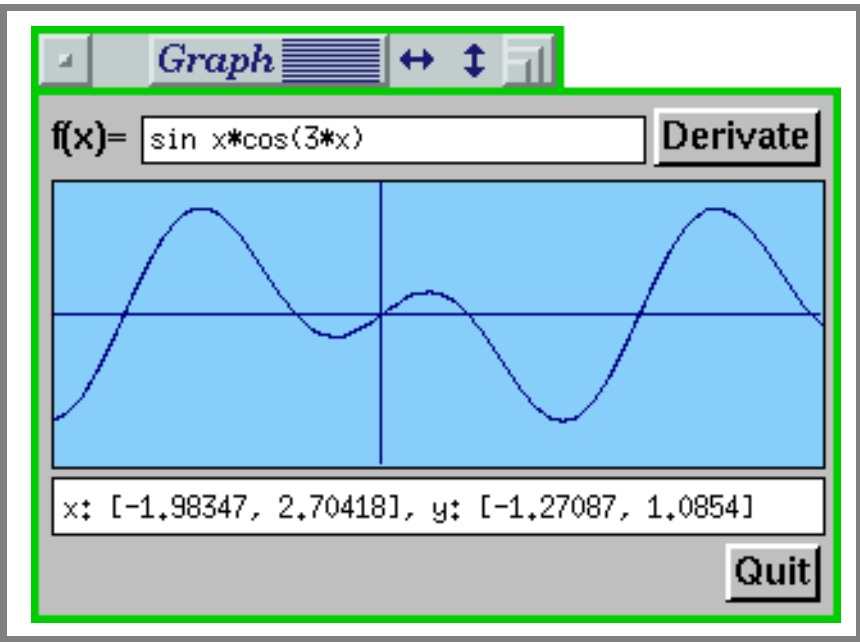
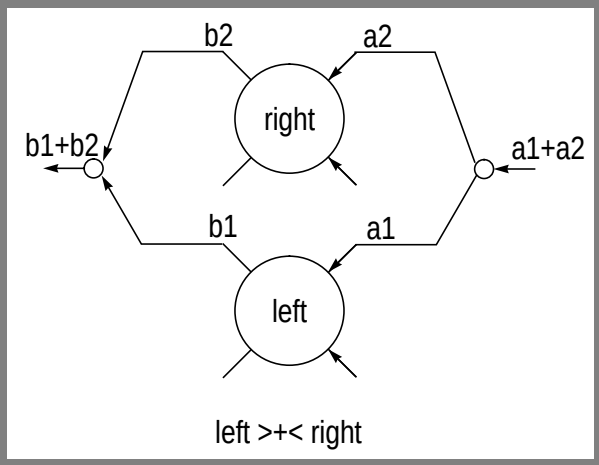
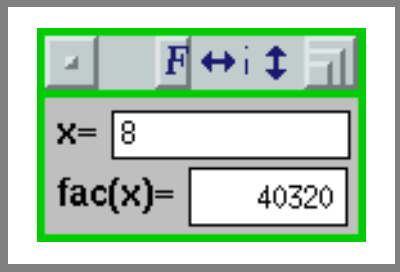
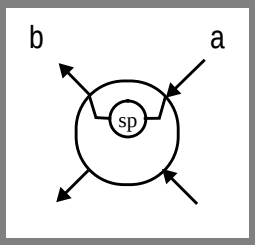
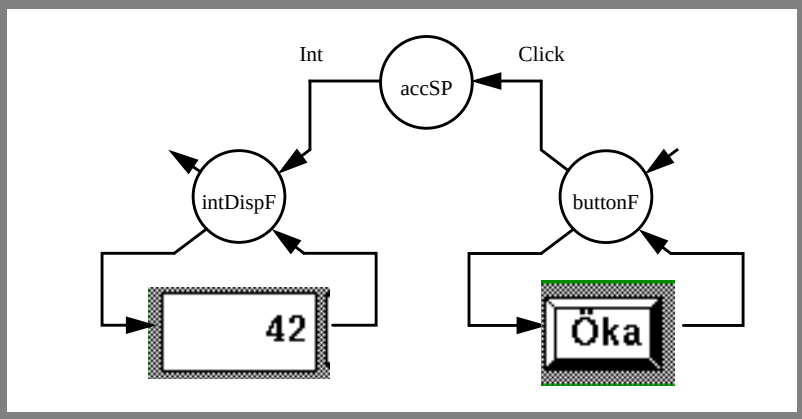
Programming with Fudgets











Magnus Carlsson & Thomas Hallgren
December 9, 1994

Table of Contents

Programming with Fudgets

A tutorial

Magnus Carlsson & Thomas Hallgren[1]

December 9, 1994

1. Introduction

In this lecture you will learn how to write programs with graphical user interfaces in Haskell. We will use the *Fudget library* which is a Graphical User Interface programming toolkit for use with Haskell and the X Windows system.

2. Graphical User Interfaces

2.1. A General Note on Graphical User Interfaces

Figure 1 illustrates a typical Graphical User Interface (GUI). There are buttons, text entry fields, popup menus, toggle buttons, etc. It is the user who decides in what order things happen. All the parts of the user interface must be prepared to respond to input at any time, independently of one another.

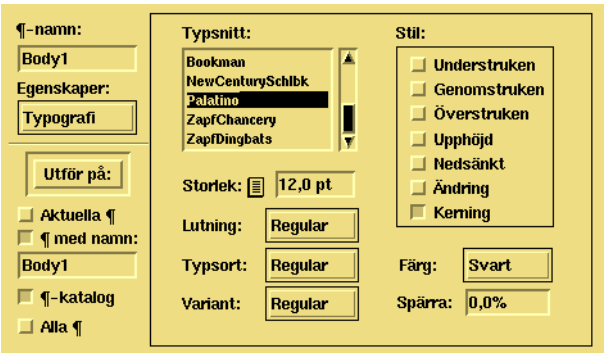
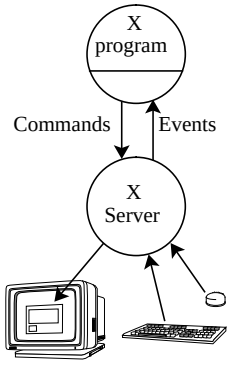


Figure 1: A typical Graphical User Interface

One important point here is that graphical user interfaces are inherently parallel. This should preferably be reflected in the programs we write to manage them. We will see later how this is accomplished with Fudgets.

2.2. GUI Programming with the X Windows system



In the X Windows system, a program interacts with the user by communicating with a server process (the X server) which handles the lowest level interface with the hardware (display, keyboard, mouse). The program sends

1. Authors' current address: Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden. Email: {magnus,hallgren}@cs.chalmers.se

a stream of *commands* (for creating windows, drawing lines, writing text etc.) to the server and receives a stream of input *events* (which tells the program about keys on the keyboard, buttons on the mouse, motion of the mouse, etc.) from the server. Most of the commands and events are related to a specific *window*. Each window has its own coordinate system used for the drawing commands. All drawing commands are relative to a window and clipped by its boundaries.

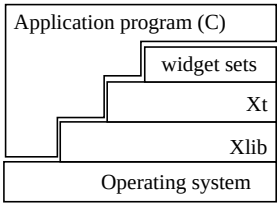
The windows have an hierarchical organization with windows in other windows. Each window has a specific position in a parent window. Most events are sent to the window under the *pointer*, which the user controls with the mouse. For each window, the programmer can decide how sensitive it should be to various events. For example, to implement a graphic button, you could create a window that is sensitive only to events telling when the pointer enters or leaves the window and when a specific mouse button is pressed or released in it. Most user interface building blocks, so called *widgets* (*wi*ndow ga*dgets*), are built up by a number of windows in this way.


A pointer

A button

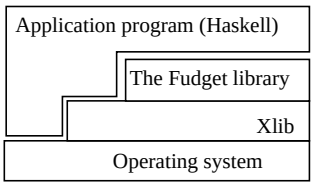
In addition to this window tree, sibling windows are organized in a *stacking order*, telling which window should hide which when they overlap. When a hidden part of a window becomes visible (e.g. because the user rearranged the windows), the X server sends an *Expose* event to the program, telling it that the newly exposed part of the window needs to be redrawn[1].

Traditionally, X programs are written in C or similar imperative languages. As illustrated by the diagram below, the interface to X goes through a set of



libraries providing different levels of abstraction. The highest level libraries provide sets of standard widgets. Typical widgets are push buttons, scroll bars, pop-up menus, etc.

The Fudgets library does not make use of the existing higher layers of libraries, but is built on top of Xlib, and provides abstractions of its own, suitable for use in a functional language. The Fudgets analogue to widgets are called *fudgets* (*fu*nctional wi*dgets*).



1. Unless you are using *backing store*, where bitmaps for the hidden parts of a window are stored off screen. This method is patented by AT&T, but allegedly, Richard Stallman implemented it way back but didn't bother to write about it.

Fudgets are similar to widgets, in that there is a set of standard fudgets, fudgets can be combined to form more complex fudgets, and when no suitable standard fudget is available, you can write one of your own. The advantage of fudgets over widgets is that the full power of a functional language (Haskell or LML) can be utilised when writing them. As a matter of fact, one of the reasons why the Fudgets library was built on top of Xlib, instead just being an interface to some existing widget set, was that we wanted to find out how good functional programs are for this kind of programming.

3. GUI Programming with Fudgets

3.1. What is a Fudget?

So, what is a Fudget? Recall that an X program essentially is a stream processor: it receives a stream of events from the X server and sends a stream of commands to it. And this is essentially what a fudget is as well: a stream processor. So, a Fudgets program (an X program written in Haskell or LML, using the Fudgets library) is a fudget, usually obtained by combining simple fudgets into more complex ones in a hierarchical structure.

What streams does a fudget process? To implement various user interface components, fudgets control windows, so they need to exchange commands and events with the X server. Different fudgets control different windows, in parallel and independently of one another. But a fudget also should be able to interact with other parts of the program. For example, selecting something from a menu should probably have some effects other than the graphical effects you see on the screen. Figure 2 illustrates the double nature of the fudget: there is a low level world, where commands are sent to a window and events returned, and then there is the high level world, where fudget specific messages are received and sent.

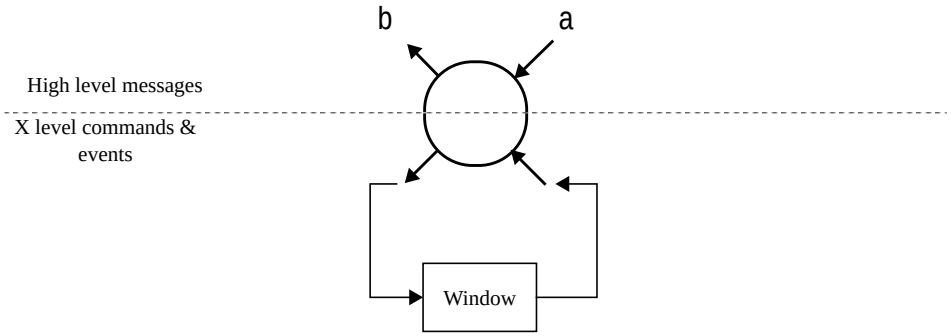


Figure 2: A fudget of type `F a b`

So, a fudget is a process with two low level streams, and two high level streams. You can think of the low level streams as always connected directly to the window controlled by the fudget. The types are the same for all fudgets. The type of high level streams varies, and what they are connected to is specified by the programmer.

In the Fudget library, the type

```
F a b
```

denotes the type of a fudget with high level input of type a and high level output of type b.

3.2. The "Hello, World!" Example

We haven't seen yet how to program the connections between fudgets, but before we continue with that, let's look at a very simple example, where no connections are required. This example illustrates what the main program should look like, as well as some other practical details.

This first program just displays a message in a window (see Figure 3).

The Fudgets library contains a fudget[1] for displaying messages,

```
displayF:: Alignment→Maybe Rect→FontName→String→F String a
```

which when given an alignment directive, an optional rectangle (position and size), a font name and last but not least, a string, will display that string properly aligned with the specified font. The display fudget can also receive new strings to display on its high level input, but we do not make use of that possibility in this program.

We have put the display in a shell window created with the shell fudget,

```
simpleShellF:: String→[WindowAttributes]→Maybe Rect→F a b→F a b
```

which given the name of the window, a list of window attributes, an optional rectangle (position and size) of the window and a fudget to put in that window, creates the shell window.

This illustrates the typical structure of a Fudgets program. In the main function, which in Haskell should have the type Dialogue,

```
type Dialogue = [Response]→[Request]
```

we call the function fudlouge,

```
fudlogue:: F a b→Dialogue
```

```
module Main where -- The "Hello, World!" program
import Fudgets

main = fudlogue (simpleShellF "Hello" [] Nothing helloF)

helloF = displayF ACenter Nothing "fixed" "Hello, World!"
```

Figure 3: The "Hello World" program

1. To be precise, displayF is a function returning a fudget, but for convenience, we will often say "a fudget" when we mean "a function returning a fudget".

which sets up the communication with the X server, gathers commands sent from all fudgets in the program and sends them to the X server, and distribute events coming from the X server to the appropriate fudgets.

3.3. Composing fudgets

As soon as we write a Fudget program with more than one user interface element, there are two things to consider:

1. How should the high level streams of the parts be connected?
2. How should the parts be placed in the window?

We start with the connections. The Fudget library contains two basic ways to compose two fudgets: parallel composition and serial composition (see Figure 4). Here are the two operators and their types:

```
>+< :: F a1 b1→ F a2 b2 → F (a1+a2) (b1+b2)
>==< :: F b c → F a b → F a c
```

Here a+b denotes the type Sum a b, which is defined as follows:

```
data Sum a b = Inl a | Inr b
```

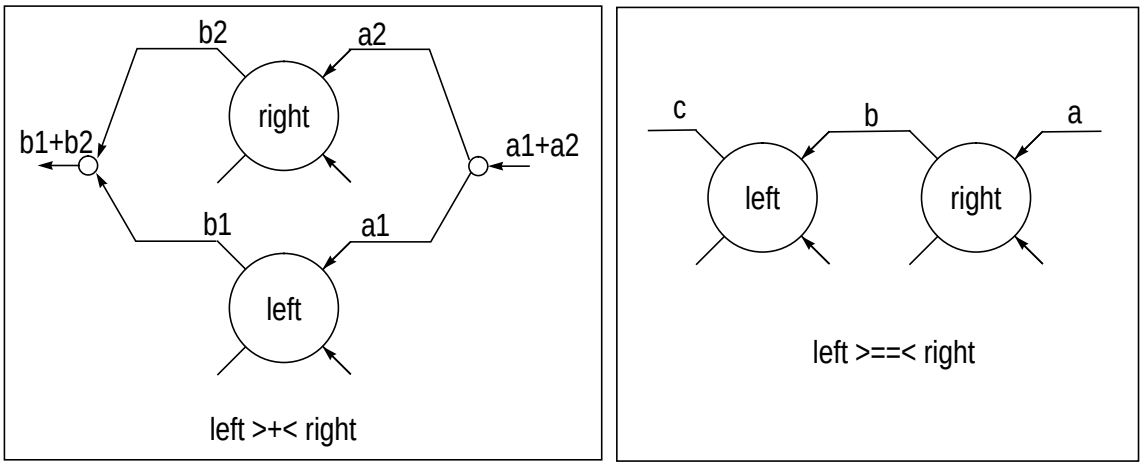


Figure 4: Parallel and serial composition of fudgets

Parallel composition allows two fudgets to operate parallel, but does not create any connections between them. Serial composition allows two fudgets to operate in parallel and in addition creates a unidirectional connection between them.

Some notes: the type of >==< is very similar to the type of the ordinary function composition operator:

```
(.) :: (b→c) → (a→b) → (a→c)
```

So, fudgets have something in common with functions.

A type like F (a1+a2) (b1+b2) can be interpreted in at least two different ways:

- as a parallel composition of two fudgets
- as a single fudget which just happens to have high level input and output streams of type a1+a2 and b1+b2 respectively.

In other words, you don't have to worry about the internal structure of a fudget when you use it. Another illustration of this is the following equality:

```
((f1>+<f2) >==< (f3>+<f4))  ==  ((f1>==<f3) >+< (f2>==<f4))
```

As a simple example of the use of a composition operator, consider modifying the "Hello, world!" program to include a quit button. There already is a ready-to-use quit button in the library,

```
quitButtonF :: F a b
```

which we can put in parallel with helloF like this:

```
main = fudlogue (simpleShellF "Hello" [] Nothing mainF)
mainF = helloF >+< quitButtonF              -- Warning: this doesn't work
helloF = ...
```

But this program doesn't work properly, because we haven't specified the layout.

3.4. Layout

Just using the combinators shown in the previous section isn't enough when combining graphical fudgets (but as we will see later, they can be used when "abstract fudgets" are involved), because you also need to specify the placement of the fudgets in the window.

Layout can be done in two ways: manually or automatically.

As we have seen above, many fudgets have an argument of type Maybe Rect. To use manual layout, you specify Just rectangle here. To use automatic layout, you specify Nothing. Automatic layout is recommended because it saves you from having to figure out sizes and pixel coordinates and because it is dynamic, i.e., the layout is automatically adjusted if the user resizes the window.

With automatic layout, the only thing you have to worry about is the relative placement of fudgets when they are combined. In the current version of the Fudget library this is done by using special versions of the composition operators:

```
>+#< :: F a1 b1→ (Int, Orientation, F a2 b2) → F (a1+a2) (b1+b2)
>==#< :: F b c → (Int, Orientation, F a b) → F a c
```

These operators work like ones without a # in the name, except that second argument should be a triple instead of just a fudget. The integer specifies the distance between the fudgets. The relative placement is specified with an element from the type Orientation:

```
data Orientation = LAbove | LBelow | LRightOf | LLeftOf
```

Now we can write a working version of the "Hello, world!" program from the previous section:

```
mainF = helloF >+#< (5, LAbove, quitButtonF)
```

You can read this as "the main fudget is the parallel composition of the hello fudget placed 5 pixels above the quit button fudget".

3.5. Adding Application Specific code

For anything but the most trivial applications, just combining fudgets from the library using various composition operators isn't enough. To create more serious applications we need ways to add application specific code that can interact with the fudgets.

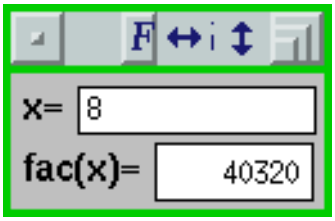


Figure 5: GUI for the FacTest program

As a first example, consider the a program for easy testing of a function, in this case the factorial function (see Figure 5). When a number is entered in the numeric entry field, its factorial will be displayed in the number display below.

In this program we need a connection from the output of the entry field to the input of the display, but we also need to apply the factorial function to each number output by the entry field before it enters the display.

The Fudget library provides two operators, which are useful in situations like this.

```
>=^< :: F b c → (a → b) → F a c          -- preprocessing input with a function
>^=< :: (b → c) → F a b → F a c          -- postprocessing output with a function
```

The resulting program is shown in Figure 6. The main fudget, mainF, is a serial composition of a numeric entry field, implemented by inIntF, and a display, outFacF. The latter inputs numbers and displays their factorials. We have used intDispF, which is similar to displayF (described in Section 3.2), and attached the factorial function fac as a preprocessor, using the operator >=^<.

```
module Main where
import Fudgets

main = fudlogue (simpleShellF "FacTst" [] Nothing mainF)
mainF = outFacF >==#< (5,LBelow,inIntF)

inIntF = "x=" `labLeftOfF`
         (inputDoneSP>^^=<intF 0 Nothing)

outFacF = "fac(x)=" `labLeftOfF`
           intDispF aRight Nothing defaultFont 0 >=^< fac

fac 0 = 1
fac n = n * fac(n-1)
```

Figure 6: The FacTest program

This program also illustrates an easy way to attach a label to a fudget. The Fudget library provides the following functions for this:

```
labAboveF, labBelowF, labLeftOfF, labRightOfF :: String → F a b → F a b
```

3.6. Stream Processors and Abstract Fudgets

Above we have seen a simple way to add application specific code to a Fudget program. But this simple method is not enough in all cases. Consider for example a program consisting of a button and a display which shows how many times the button has been pressed. The button fudget in the library just outputs a Click when it is pressed, so we need application specific code that increments a counter and outputs its value to the display whenever a the button outputs a click (see Figure 7).

Figure 7: The Small Counter

This problem is solved by letting the application specific code be stream processors, just like fudgets. The application specific stream processors are called *abstract fudgets*, since they can be thought of as fudgets without any physical appearance. You obtain an abstract fudget by applying the function absF to a *stream processor*

```
absF :: SP a b → F a b
```

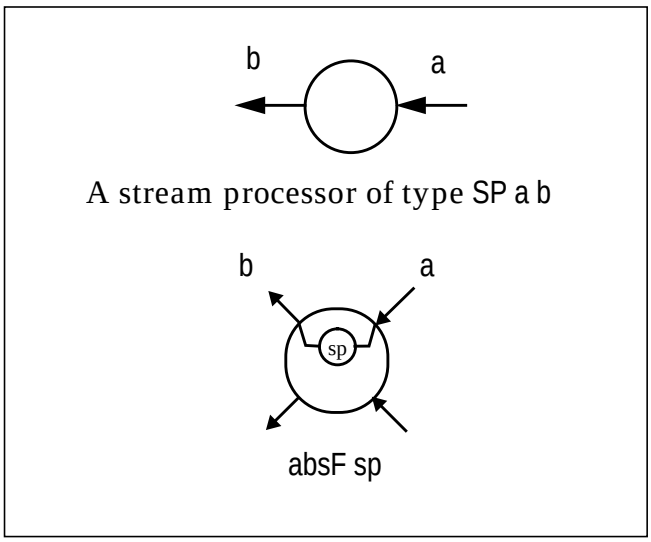


Figure 8: Stream Processors and Abstract Fudgets

Then the question is: how do you write stream processors? The fudget library provides three basic functions to construct stream processors:

```
nullSP :: SP a b                    -- a "dead" stream processor
putSP :: [b] → SP a b → SP a b      -- writes to the output stream
```

```
getSP :: (a → SP a b) → SP a b          -- reads from the input stream
```

The library also provides a number convenient stream processor constructors that can be derived from the basic ones. One of them is mapSP:

```
mapSP :: (a → b) → SP a b
```

The two operators >=^< and >^=< from the previous section can be defined using mapSP and absF in the following way:

```
fud >=^< f == fud >==< absF (mapSP f)
f >^=< fud == absF (mapSP f) >==< fud
```

For the small counter program we need a click counting stream processor to connect between the button and the display (see Figure 9).

Figure 10 shows the program for the small counter. The click counting stream processor is called accSP. It uses a recursive auxiliary function accSP', which maintains the state (the number of clicks so far) in the parameter n.

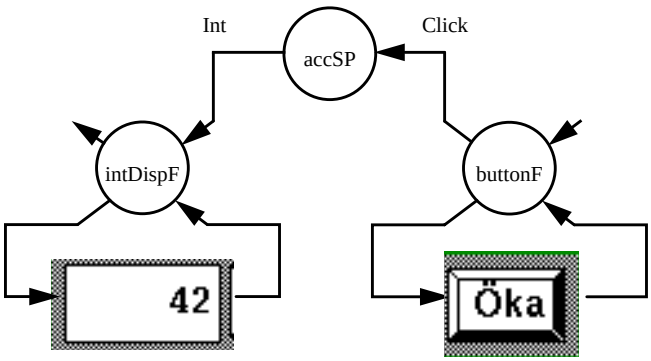


Figure 9: Circuit diagram for the small counter

```
module Main(main) where -- The Small Counter Example
import Fudgets

main = fudlogue (simpleShellF "Räknare" [] Nothing counterF)

counterF= (intDispF aRight Nothing "fixed" 0 >==< absF accSP)
                >==#<
          (5,LLeftOf,buttonF Nothing buttonFont [] "Öka")

accSP = accSP' 0
   where accSP' n =
            getSP $ \ Click →
            putSP [n+1] $
            accSP' (n+1)
```

Figure 10: The Small Counter Program

4. A bigger example

In this section we take a look at a slightly bigger Fudget program. The program is thoroughly explained. Some new useful operators are introduced, but most of the program resembles what we have already seen in the earlier sections.

4.1. The Counter Example

This example illustrates how to connect several fudgets together with automatic layout and how to attach application specific code using a stream processor. The program is a simple counter, consisting of a display showing the current value of the counter, and buttons to increment the counter, reset the counter and to quit the program (see Figure 11).

Let's examine the structure of this program, starting from the top. In the function main, the main window is created with simpleShellF. The size argument is Nothing, which means that the size of the shell window is determined by the dynamic layout mechanism. Initially, the window will be just large enough to fit the fudgets contained in it, but the user is allowed to resize the window, in which case the contents of the window is adjusted to fit the new size.

The contents of the shell window is the counter fudget, counter, which is defined as a serial composition of a display, an "accumulator", and some buttons. The idea is that when the a button is pressed, a message is sent to the accumulator, which computes the new value of the counter and sends it to the display. The accumulator is a stream processor, attached as a preprocessor to the display fudget. This is the purpose of the >=^^< operator, which has the following definition:

```
(>=^^<):: F b c→ SP a b →F a c
fud >=^^< sp = fud >==< absF sp
```

So, counter consists of two fudgets: the display, which has its input preprocessed by the accumulator, and the buttons. The operator >==#< connects the output of the buttons to the input of the display. It also determines the relative placement of the display and the buttons. The display is placed above the buttons, with sep (i.e. 5) pixels separating them.

The display itself is implemented with intDispF, taken from the Fudgets library. The argument aRight makes the integer displayed right adjusted, Nothing makes the position and size of the display determined by the dynamic layout system, font tells which font to use, and startstate is the integer displayed before any other numbers are received on the input.

The buttons are implemented as a list of buttons, combined to one fudget with untaggedListLF. The three buttons are all created with the library fudget buttonF, and all have the same appearance, except for the label in them, and we use the shorthand b to capture this. The behaviour of buttonF is to send the message Click whenever the button is clicked with the mouse. We want our buttons to do different things, so we attach post-processors to the buttons, to convert the clicks to other messages.

The first two buttons are similar: they both perform some operation on the value of the counter. These button sends messages to the display. The messages contain the actual function to be performed on the counter. The shorthand bf captures what they have in common. The argument s is the button label, and f is the function to be sent to the display. const f is attached as a post-processor to the button itself, to replace the Clicks by the function. This is accomplished with the >^=< operator, which allows you to specify a function to be applied to all elements in the output stream.

Clicking the quit button should terminate the program. This is achieved by feeding the Clicks from the quit button to the special quit fudget, quitF, which has the special property that feeding a message to it causes the program to

```
module  Main(main) where -- A very simple calculator
import Fudgets

main :: Dialogue
main = fudlogue (simpleShellF "Counter" [] Nothing counter)

counter :: F (Int,a) b
counter = (display >=^^< acc) >==#< (sep,LAbove,buttons)

display :: F Int b
display = intDispF aRight Nothing defaultFont startstate

type State = Int
type StateModifier = Int -> Int

buttons :: F (Int,a) StateModifier
buttons =
  let b s = buttonF Nothing buttonFont [] s
      bf s f = const f >^=< b s
      bq s   = quitF   >==< b s

      buttonlist = [bf "Add 1" (+1),
                    bf "Clear" (const 0),
                    bq "Quit"]

      layout = horizontalL sep
  in untaggedListLF layout buttonlist

acc :: SP StateModifier State
acc =
  let transform state =
        getSP $ \op ->
           let state' = op state
           in putSP [state'] $ transform state'
  in transform startstate

startstate = 0

sep = 5 -- separation, space between buttons
```

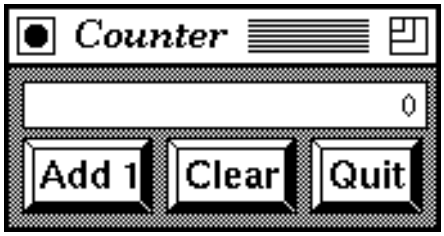


Figure 11: The Counter Example

terminate. quitF does not have an associated window (although it isn't really an abstract fudget). In the shorthand bq, the operator >==< is used to connect the output of a button to the input of the quit fudget.

The three buttons are enumerated in buttonlist, which is then the main argument to untaggedListLF, which with its other argument, layout, is instructed to place the buttons in buttonlist in a horizontal row with sep pixels between them. It also merges the output streams from the buttons into one stream, without tagging the messages. (There is a similar fudget combinator, listLF, which also tags the messages, so that you easily can tell messages from different sub-fudgets apart, and also direct input to specific sub-fudgets.)

The remainder of the program contains some auxiliary definitions, most notably the stream processor acc. It repeatedly reads a state modifier from its input stream (using getSP), applies it to the internally maintained state and then outputs the new state (using putSP).

Exercises:

1. Extend the counter to a pocket calculator. You will probably need to change the State type. You may find the layout function matrixL useful.

4.2. How should Fudget Programs be structured?

The counter program above illustrates one particular way to structure Fudgets programs. We have kept the buttons and the operations they perform together. The advantage with this structure is that it is very easy to add new buttons to the program. To add a decrement button, for example, just include

```
bf "Sub 1" (\x→x-1)
```

at the appropriate position in buttonlist.

By directly giving the function that manipulate the state maintained in the accumulator, we have hard coded the fact that the type of the state is Int. This of course makes it hard to add buttons with functions that require a more complex state. Making the state an abstract data type would make the program easier to extend in this way.

Another way to structure a Fudgets program is to keep the user interface parts and the data processing parts separate. This makes it easier to change the user interface without changing the rest of the program, or even to have several completely different user interfaces, for example to allow you to run a program both from an old fashioned text terminal and from a modern work-station running X Windows.

Exercises:

2. Restructure the counter program (or your pocket calculator) so that it can be used both with an old fashioned text interface and a Fudget based graphical interface. The function

   ```
   concRunSP :: SP a b → [a] → [b]
   ```

   can be used to create a text based interface with a stream processor.

5. Writing Your Own Fudget

A Fudgets program is a hierarchical structure of fudgets. The fudgets at the bottom level are not built by composing existing fudgets, but by applying windowF to a *fudget kernel*.

```
windowF :: [Command] → Maybe Rect → K a b → F a b
```

K a b is the type of a fudget kernel.

```
type K a b = SP (Message Event a)  (Message Command b)
type Message low high = Low low | High high
```

As you can see, fudget kernels are, like abstract fudgets, stream processors. The input and output messages can be either low level messages, i.e., events and commands for communication with the X Windows system, or high level messages for communication with other fudgets.

Below, we illustrate by an example what a typical fudget kernel looks like.

5.1. The Dial Example

In this example we write a fudget called dialF, which implements a simple dial. Its type is:

```
dialF: Maybe Size → Double → F Double a
```

Given an optional size and an initial value to display, dialF creates a fudget that for every floating point number received on the high level input, changes the dial to show this value. It does not produce any high level output. The input number specifies the angle of the hand measured clockwise in units of complete turns. 0.0 is straight up.

The Haskell implementation of dialF is shown in Figure 12. It is implemented by applying windowF to the fudget kernel dialK and some other arguments.

The first argument to windowF is a list of commands to be executed right after the window is created. In this case, the list contains two commands. The first one sets the event mask. Since this fudget is output-only, the only events we are interested in are exposure events. They are generated by the X server when the contents of a window need to be re-drawn because the window becomes exposed after being obscured by another window. A fudget that responds to user input would in addition include things like ButtonPressMask, ButtonReleaseMask, PointerMotionMask, KeyPressMask, etc.

The second of the two initial commands is the LayoutLimits pseudo-command. With the argument Layout size False False, we tell an enclosing fudget that the initial size of the dial fudget should be at least size, and that neither the horizontal nor the vertical size should be fixed, but can vary freely. We use the size supplied in optsize, or (somewhat arbitrarily) pick the size (70,70) if optsize is Nothing.

The second argument to windowF is an optional rectangle that determines the position and size of the window. Here we supply Nothing, which means that we want the dynamic layout system to compute the position and size. If you specify some rectangle here, you indicate that this fudget should not be part

```
module DialF(dialF) where
import Fudgets

dialF :: Maybe Size -> Double -> F Double a
dialF optsize startvalue =
  let startcmds = [ChangeWindowAttributes [CWEventMask [ExposureMask]],
                   LayoutLimits (Layout size False False)]
      size = case optsize of
               Just s -> s
               Nothing -> Point 70 70
 in windowF startcmds Nothing (dialK startvalue size)

dialK startvalue size =
    wCreateGC RootGC [GCLineWidth (Width 3)] (\gc ->
      let redraw v s = putK (drawDial gc v s) (dialK' v s)
          dialK' value size =
              getK (\msg->
                case msg of
                  Low (Expose _ 0) -> redraw value size
                  Low (LayoutSize size') -> redraw value size'
                  High value' -> redraw value' size
                  _ -> dialK' value size)
      in dialK' startvalue size)

drawDial gc value size =
  let r = scalePoint 0.5 size
      p = r `padd` rect r (2.0 * pi * value)
  in map Low [ClearWindow, WDrawLine gc (Line r p)]

rect (Point w h) r =
  Point (floor (fromInt w * sin r)) (floor (fromInt (-h) * cos r))
```

Figure 12: The Haskell implementation of the dial fudget

of the dynamic layout system, but instead have the fixed placement given by the rectangle. In this case the fudget should not output a LayoutLimits command.

All drawing commands in Xlib take a GC as an argument. GCs, graphic contexts, contain many parameters that control how the drawing is done, such as foreground and background colours, line width for line drawing, which font to use for text, etc. The Fudgets library provides the function wCreateGC,

wCreateGC: GCId → GCAttributeList→ (tGCId→K a b) → K a b

which takes a template GC, a list of GC attributes to modify the template with, a fudget kernel parameterized by a GC, and returns the argument fudget kernel applied to a newly created GC. As the template GC you can use any GC you have already created, or RootGC, which is a pseudo GC that can be used only as a template and not for drawing.

In our fudget kernel, dialK, we create a GC with RootGC as a template, and change the line width to 3 pixels.

After creating the GC, dialK turns into a state machine, described by the recursive function dialK'. The state is held in the two arguments of dialK': value and size.

A fudget kernel normally uses putK and getK to input/output messages.

```
putK :: [b] → K a b → K a b
getK :: (a → K a b) → K a b
```

They are actually the same functions as putSP and getSP, but the automatically inferred types becomes much more readable when they contain K a b instead of SP (Message Event a) (Message Command b)...

Fudgets that show something in a window must be prepared to re-draw what they show when they receive an exposure event. In Fudgets, exposure events take the form Expose rect n, where rect tells what part of the window that actually needs to be re-drawn. Exposure events generally come in bursts, and n tells how many more exposure events containing further rectangles will follow after this event.

The dial is very easy to draw, so keeping track of which part of the window to redraw will not significantly reduce the amount of re-drawing required. It is enough to redraw the entire dial after receiving the last exposure event in a burst.

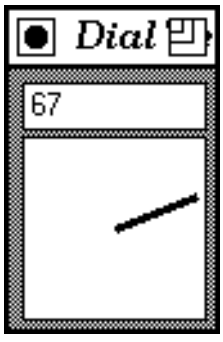
If dialK' receives the pseudo-event LayoutSize, the dynamic layout mechanism has resized the window. dialK' responds by redrawing the dial in its new size.

If dialK' receives a new value in a high level message, the dial is redrawn with the new value.

Input other that exposure events, the LayoutSize event, high level messages is silently ignored. A fudget is in general expected to ignore any low level input it doesn't know how to handle.

After dialK follows two support functions: drawDial, that generates the drawing commands for drawing the dial, and rect, which converts coordinates from polar to rectangular form.

Figure 13 shows a simple program to test dialF:




```
module Main where
import Fudgets
import DialF

main = fudlogue (simpleShellF "Dial" [] Nothing tstF)

tstF = dialF Nothing 0.0 >=^< (\x->fromInt x/360.0) >==#<
       (4,LBelow,inputDoneSP>^^=<intF 0 Nothing)
```

Figure 13: Test program for dialF

Exercises:

3. Add drawing commands to make the dial look nicer.
4. Enhance dialF to an input/output fudget. Allow the user to change the value of the dial by clicking the mouse in it. Use GrabButton and motion events to make the dial follow the mouse while a mouse button is pressed.

6. Compiling and Getting Documentation

6.1. Compiling Fudget Programs

To compile a Fudgets program, use the command hbcxmake. For example, if your main program is in a file called Counter.hs, then you can compile it with the following command:

```
hbcxmake Counter
```

hbcxmake is an automatic make program, so even if your program consists of more than one module you only need to run hbcxmake on the main module.

6.2. On-line documentation

- Information on the Fudgets system can be found in the WWW via URL

  http://www.cs.chalmers.se/Fudgets/

  (Use a WWW browser like netscape or Mosaic). From there you can find a link to the reference manual and other documentation.
- The command fudgrep can be used to find types of fudgets, etc. (This is just grep in the Fudgets.hi, the interface file for the Fudgets library.)
- Some examples and demos can be found in

  /usr/src/cs/local/Fudgets/Examples/
  /usr/src/cs/local/Fudgets/Demos/

6.3. Note!

Be aware that the Fudget library is not a finished product! Some things you expect to find may be missing, or may not work as you expect.

- Tell us if you think something is strange or missing (e.g. by email to hallgren@cs.chalmers.se)

7. Summary

Here is a brief list of useful things from the Fudget library.

- Top level

  ```
  fudlogue:: F a b→Dialogue
  simpleShellF:: String→[WindowAttributes]→Maybe Size→F a b→F a b
  ```

- Fudgets (User Interface Elements)

```
displayF:: Alignment→Maybe Rect→FontName→String→F String a
intDispF:: Alignment→Maybe Rect→FontName→Int→F Int a
buttonF :: Maybe Rect → FontName → [(ModState, KeySym)] → String → F a Click
quitF, quitButtonF :: F a b
intF :: Int → Maybe Rect → F Int (InputMsg Int)
stringF :: String → Maybe Rect → F String (InputMsg String)
inputDoneSP :: SP (InputMsg a) a
labAboveF, labBelowF, labLeftOfF, labRightOfF :: String → F a b → F a b
moreF, menuF, radioF, toggleButtonF
```

- Fudget combinators

```
>==<, >==#<                                  serial composition without/with layout
>+<, >+#<                                    parallel composition
>+#< :: F a1 b1→ (Int, Orientation, F a2 b2) → F (a1+a2) (b1+b2)
>==#< :: F b c → (Int, Orientation, F a b) → F a c

listLF :: Layouter → [(t, F a b)] → F (t, a) (t, b)
untaggedListLF :: Layouter → [F a b] → F (Int, a) b
                                             parallel composition with layout
```

- Layout

```
aRight, aLeft, aTop, aBottom, aCenter :: Alignment
data Orientation = LAbove | LBelow | LLeftOf | LRightOf
data LayoutDir = Horizontal | Vertical
horizontalL, verticalL :: Int → Layouter
matrixL :: Int → LayoutDir → Int → Layouter
```

- Attaching application specific code

```
absF :: SP a b → F a b                       abstract fudget
>^^=<, >=^^<                                attaching post/preprocessors
>^=<, >=^<                                   attaching functions
```

- Stream processors

```
nullSP :: SP a b                             dead stream processor
putSP :: [b] → SP a b → SP a b               writes to the output stream
getSP :: (a→SP a b) → SP a b                 reads from the input stream
mapSP, concatMapSP, mapAccumlSP
concatSP, zipSP
```

- Fudget Kernel Programming

```
windowF :: [Command] → Maybe Rect → K a b → F a b
putK, getK                                   like putSP, getSP
wCreateGC :: GCId → GCAttributeList → (GCId → K a b) → K a b
changeBg :: ColorName → K a b → K a b
safeLoadQueryFont :: FontName → (FontStruct → K a b) → K a b
allocNamedColor :: ColormapId → ColorName → (Color → K a b) → K a b
defaultColormap :: ColormapId
font_id :: FontStruct → FontId
wDrawString, wDrawLine, wDrawRectangle, wFillRectangle, wDrawArc, ...
```

- Types

data SP a b	abstract type for stream processors
type F a b, type K a b	the Fudget type, Fudget kernels
data Message a b = Low a | High b	
data Sum a b = Inl a | Inr b	
data Command = ...	subset of Xlib commands + some others
data Event = ...	subset of X events + some others
data DrawCommand = ...	drawing commands
data EventMask = ButtonPressMask | ExposureMask | ...	
data GCAttribute color font = GCLineWidth With | GCForeground color | GC font | ...	
data Maybe a = Nothing | Justa	
type FontName = String	
type ColorName = String	
data InputMsg a = InputChange a | InputDone KeySym a	
data Point = Point Int Int	
type Size = Point	
data Line = Line Point Point	
data Rect = Rect Point Size	

- Utilities

defaultFont, buttonFont, menuFont	can be changed from the command line
args, argKey	to read command line arguments
padd, psub :: Point→Point→Point	vector arithmetic
xcoord, ycoord :: Point -> Int	selectors