Implementing Fudgets with Standard Widget Sets

Alastair Reid & Satnam Singh*
Computing Science Department
University of Glasgow

July 6, 1998

Abstract

Carlsson and Hallgren [1] describe the implementation of a set of "functional widgets" (Fudgets): components for programming graphical user interfaces under the X window system using the non-strict functional programming language Haskell. We describe an alternative implementation based on existing widget sets (currently Openlook and Motif). Our purpose is twofold: to show that the Fudgets approach can be applied to existing widget sets; and to discuss problems experienced with Fudgets during an industrial case study.

1 Introduction

Imperative language programmers enjoy relatively easy access to the graphics resources of workstations. The graphics hardware is manipulated by side-effecting procedure calls. Even if the library of graphics procedures is written in one imperative language (e.g 'C'), programs written in another imperative language can usually make calls to foreign procedures. For example, Ada allows foreign procedures to be called by giving a standard pragma. Ada compilers also allow Ada routines to by called by alien procedures.

This report describes a library for building high quality user-interfaces for the purely functional lazy programming language Haskell. Graphics operations are produced by making alien procedure calls to C language routines. Communicating data between Haskell and C programs is not trivial because Haskell is a lazy language, has a garbage collector and uses a very different representation for data (even for simple types like integers). We outline how to write Haskell programs that communicate data with C routines in an orderly fashion using the Glasgow IO monad.

The style of our interface is deliberately similar to the idiomatic style used in C for writing X Windows graphics software. This invites comparison with equivalent C programs and makes it easier to use the extensive body of X11 programming manuals.

The structuring technique employed is based on the excellent Fudgets systems which uses higher-order combinators to glue together collections of user interface components. We describe some of the problems that arise from the static nature of the user interfaces generated by the Fudgets system. The Fudgets system defines its own user interface components. We also show how the Fudgets approach can be modified to use existing user interface components. In particular, we have adapted Fudgets to use OpenLook and Motif for building commercial quality and standardised user interfaces.

*Email: {areid,satnam}@@dcs.glasgow.ac.uk

2 C Programmer's view of X widgets

The target graphics system for our graphics library is the X Window System. This system runs on a large variety of graphics workstations and affords us some degree of device independence.

The X Window System is based around a server-client model. A client program (e.g. a drawing program) need not run on the machine that actually supports the display (a graphics workstation). Indeed, the client machine may have no display at all because the client and server are connected over a network. A client program sends requests to the server to draw lines, points etc. The client program is also notified about events on the server's display.

At the lowest level, the X Window System is a network protocol which provides a network transparent interface for servers and clients. A C program language interface to this protocol is called Xlib. This provides data types and procedures for performing very basic graphics operations. Xlib is usually the lowest level at which X11 applications are written. However, little support is provided for building user interfaces comprising of components like buttons, menus and scrollbars.

The X Intrinsics Toolkit (Xt) is a collection of C types and procedures that describe the infrastructure need to build graphical user interfaces. A mechanism is provided for creating user interface components called *widgets*. Composite widgets may contain other widgets, allowing user interfaces to be constructed in a modular fashion as a widget tree. Widgets contain local state and are often implemented as finite state machines. The system we describe uses Xlib and Xt.

Xt does not define the behaviour or appearance of any particular widget. It only provides a 'backplane' into which specific *widget sets* can be plugged into. Widget sets include Athena (distributed with Xt), OpenLook Intrinsic Toolkit (OLIT) and Motif. Though broadly similar, different widget sets have different resources and callbacks, so it is hard to modify a program written with one widget set to work with another widget set.

The Xt system for managing events like button clicks and menu selection is based around *callbacks*. Callbacks are similar to interrupts. A widget can have several kinds of events. For each widget and each kind of event, a callback routine can be specified. (For example, a button would have a handler that is called whenever it is clicked. The callback routine is like a closure (it is basically a code-environment pair). Unlike an interrupt, the client program is not immediately interrupted and control transfered to the handler. Instead, this event is queued. The top level of an Xt program contains a loop that waits for an event and then dispatches the event by calling the appropriate callback. Thus, only one callback can occur at any time. The rest of the program is held up until the callback routine has finished.

The execution of Xt programs takes place in three distinct phases. First a connection to the X server is created. Once the connection is formed, a widget that corresponds to the root window of the server display is returned. A client has as its top level a *shell* widget whose parent is the root window. The widgets of the client program are realised and the client program enters its event loop.

Each widget has associated with it a set of *resources*. Resources allow certain aspects about the behaviour or appearance of a widget to be determined either when it is initialised or during execution. Resources are also useful for describing the positioning of widgets on the screen, colouration or internationalisation. Both X11 and Xt are equipped with sophisticated resource database managers.

To illustrate the idiomatic C style for writing X11 software, we show below a (slightly simplified) program that changes the label text of a user interface component (taken from [1]):

```
static int count = 0;

static void setDisplay(Widget display, int i)
{
```

```
  char s[10];
  Arg wargs[1];

  sprintf(s, "%d", i);
  XtSetArg(wargs[0],XmNlabelString, s);
  XtSetValues(display, wargs, 1);
}

static void increment(Widget display)
{
  count++;
  setDisplay(display, count);
}

void main()
{
  Widget top, row, button, counter;

  top = XtInitialise();
  row = XmCreateRowColumn("row", top);
  display = XmCreateLabel("display",row);
  button = XmCreatePushButton("button",row);
  setDisplay( display, count );
  XtAddCallback(button, increment, display);
  XtRealizeWidget();
  XtMainLoop();
}
```

The main procedure sets up a connection to the X11 server and creates a hierarchy of widgets. A callback routine is declared for the button, namely `increment`. Whenever the button is clicked, an event is registered. The `XtMainLoop` procedure processes this event by looking up and then executing the callback declared for the button (i.e. `increment`). The `increment` routine simply updates a global counter variable and then makes the label display the decimal representation of this count as its label text. The label text is modified by updating the label resource (`XmNlabelString`) for the button widget.

3 Accessing widgets from Haskell

Our method of accessing the various X and widget libraries might be regarded as the most straightforward approach: for every library function that we want to access, we define a Haskell function that calls that function.

Since the X-library functions have various side-effects (the most obvious of which is drawing an image on the screen) it is necessary to ensure that the operations occur in the correct sequence. Previous approaches [8, 7] have guaranteed that actions occur in a strict sequence by sending a list of commands to an interpreter (written in an imperative language) which executes the commands in the order they are received.

A more recent approach (supported by the Glasgow compiler) is to use a *monad* [4] to execute a series of side-effecting actions in a strict sequence. Briefly, the *Glasgow IO monad* provides:

- A data type `IO` α which is the type of a (possibly side-effecting) action which, when executed, returns a value of type α.

- A mechanism that allows arbitrary code written in an imperative language to be used as an action of type `IO` α.
- A function `returnIO ::` $\alpha \rightarrow$ `IO` α which, when executed returns its argument.
- The combinator `thenIO ::` `IO` α `->` (α `->` `IO` β) `->` `IO` β which combines two actions into one action. When executed, `a1 ‘thenIO‘ a2` first executes `a1` obtaining a result `r` and then executes the action `a2 r`.

We refer the reader to [4] for further details.

Using the monadic approach, the main task of providing access to a set of imperative library functions is to define a set of Haskell functions which call the corresponding imperative function. The major difficulty here is in passing values from Haskell into the imperative functions and from imperative functions into Haskell. For simple values such as integers and strings, we were able to use the method of "unboxing" described by Peyton Jones and Launchbury [3]; to allow us to pass more complex values such as callbacks, we made a small, general-purpose extension to the Glasgow compiler.[1]

This basic approach can also be used to translate programs which use the X and widget libraries into Haskell. One further difficulty lies in the implementation of global variables. We use the following solution described by Launchbury in [5].

- The type `Var` α is an abstract data type of mutable variables of type α.
- Given an initial value `x` say, executing the operation `newVar x` allocates a variable with initial value `x`, and returns a reference to the variable.
- Given a variable `v::Var` α, executing `readVar v` reads the current value of the variable `v`. Similarily, executing `writeVar` updates the value of the variable.

For example, the program at the end of section 2 may be "translated" into the following Haskell program.

```
>  increment :: Label d => Var Int -> d -> IO ()
>  increment var display =
>    readVar var                          ‘thenIO‘ \ count ->
>    writeVar var (count + 1)             ‘thenIO‘ \ _ ->
>    setDisplay display (count + 1)
>
>  setDisplay :: Label d => d -> Int -> IO ()
>  setDisplay display count =
>    setLabel display (LabelString (show count))
>
>  mainIO :: IO ()
>  mainIO =
>    initialise "Xtest"                   ‘thenIO‘ \ top ->
>    createRowColumn "row" top            ‘thenIO‘ \ row ->
```

[1] Our initial implementation used a different approach based on the fact that callbacks are only called by the event loop and it is possible to write your own eventloop for X. All we had to do was write callback routines which insert "callback events" into an event queue and replace the event loop with a Haskell loop which repeatedly calls the normal event-handling routines (which might cause callbacks to happen) and then dispatches any callbacks found in the event queue. Since the event loop is written in Haskell, there is no difficulty in calling callback routines written in Haskell.
This approach could be used quite effectively (and efficiently!) by those wishing to apply our overall approach under other compilers.

```
>    createLabel "label" row              `thenIO` \ display ->
>    createButton "Press Me!" row         `thenIO` \ button ->
>    newVar 0                             `thenIO` \ countVar ->
>    setDisplay display 0                 `thenIO` \ _ ->
>    addButtonCallback button
>            (increment countVar display) `thenIO` \ _ ->
>    realizeWidget top                    `thenIO` \ _ ->
>    mainloop
```

The above example illustrates how one might (naively) write GUI's in Haskell: first write the program in C and then translate it into Haskell. However, even if with practice we learn to avoid writing the program in C first, this kind of approach cannot be expected to lead to functional GUIs which are any simpler than their imperative counterparts. The next section discusses an approach which is dramatically simpler than the above.

4 Fudgets

In [1] Carlsson and Hallgren argue that functional languages are better for implementing GUIs because they offer better abstraction facilities. In particular, their approach makes extensive use of higher-order functions to capture common patterns of coding within GUI programs.

The essence of Carlsson and Hallgren's approach is to treat each component of the user-interface as a "black box" (a *Fudget*) receiving input on a single "input pin" and sending output on a single "output pin."

In Carlsson and Hallgren's implementation (see figure 1a), each (primitive) fudget is responsible for (at most) one window whose appearance it controls by sending X-protocol requests to the X-server and which communicates with the fudget by sending X-events to the fudget.

In our implementation (see figure 1b), each (primitive) fudget is responsible for (at most) one widget whose appearance and behaviour is controlled by calling resource setting routines (such as `setLabel`) and which communicates with the fudget by executing callbacks.

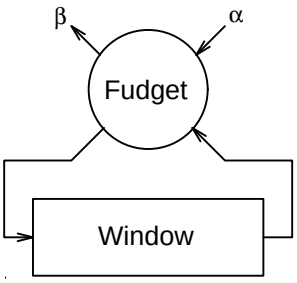

Figure 1a. A Swedish Fudget

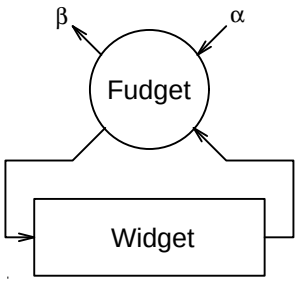

Figure 1b. A Glasgow Fudget

From the programmer's point of view, there is little difference between the two approaches.

Some examples of simple fudgets are:

- `button :: String -> F` α `Click` encapsulates the pushButton widget. The `String` is used as the label displayed on the button. When the user clicks on the button, a value `Click`[2] is sent to the output pin. (All input is ignored.)

[2] In Haskell, the type `Click` is defined by `data Click = Click`.

- `label :: Text` α `=> F` α β encapsulates the label widget used for outputting (small) pieces of text. When a value is received on its input pin, its textual representation is displayed on the label widget. (No output is produced.)
- `textField :: F` α `String` encapsulates the text field widget used for inputting (small) pieces of text. When a value is received on its input pin, the current text entered by the user is sent to the output.

It is also useful to create fudgets which are not associated with any widgets at all. Two such fudgets are:

- `ioToFudget ::` $(\alpha \rightarrow$ `IO` $\beta) \rightarrow$ `(F` α β`)` encapsulates an `IO` operation. When a value is received on its input pin, the `IO` operation is applied to that value (and executed) and the result is sent to the output pin.

  A typical use of this function is to write the input text to a file or to perform a database transaction on receiving data on the input pin.
- `stateMachine ::` $((s, \alpha) \rightarrow (s, \beta)) \rightarrow s \rightarrow$ `F` α β encapsulates a piece of local state. When a value is received on its input pin, the input and current state are used to calculate an output and a successor state and the output is sent to the output pin.

The strength of Carlsson and Hallgren's approach lies in the provision of fudget combinators which allow simple fudgets to be combined into more powerful combinators. For example fudget composition is achieved with the combinator `<==<` `::` `F` β $\gamma \rightarrow$ `F` α $\beta \rightarrow$ `F` α γ which connects the output of the second fudget to the input of the first fudget (see figure 2). Like function composition, fudget composition is associative.

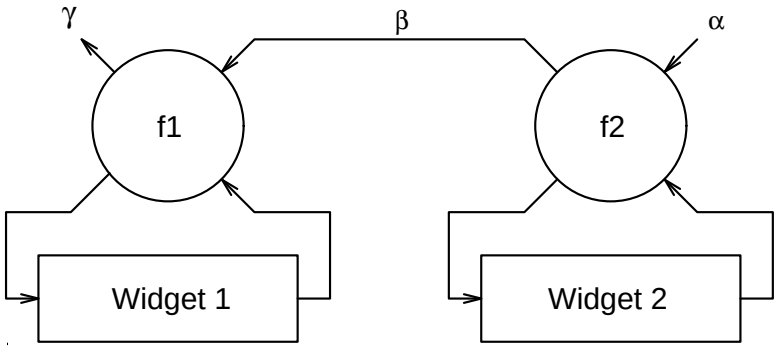

Figure 2. The fudget `f1 <==< f2`

For example, the example discussed in the previous sections can be implemented as follows (see figure 3):[3]

```
> mainIO :: IO ()
> mainIO = doFudget counter
>
> counter = label 0                 <==<
>           stateMachine count 0    <==<
>           button "Press Me!"
>  where
>   count (c, Click) = let c' = c+1 in (c',c')
```

[3] The function `doFudget :: F` α $\beta \rightarrow$ `IO ()` initialises the widgets contained within a fudget and enters the event loop.

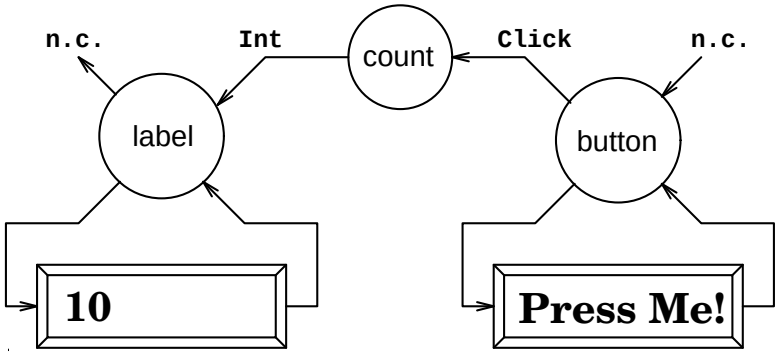

Figure 3. The Counter Fudget

5 Implementation of Fudgets

It is straightforward to implement fudgets using the library discussed in section 3. Our implementation of fudgets is based on the following observations:

- When a fudget is created, we must call the creation function to create the corresponding widget. All creation functions have a `parent` parameter which is used by X-toolkit to create the widget hierarchy. Therefore a fudget must be a function taking (at least) a parent widget as a parameter.
- Since widgets communicate with fudgets by executing callbacks, the simplest way for fudgets to communicate with each other is by executing functions of the same type as callbacks. We call such functions "handlers."

  `type Handler` α `=` $\alpha \rightarrow$ `IO ()`

  That is, when a fudget is created, it is passed an output handler (which it will call when it wants to send output) and returns an input handler (which is called when it is being sent input).

For these reasons, the type `F` α β is defined by:

`type F` α β `= Widget` $\rightarrow$ `Handler` β $\rightarrow$ `IO (Handler` α`)`

The definition of the four fudgets and combinators used in the above example is straightforward:

- The `button` fudget creates a pushbutton widget; adds a callback; and returns an input handler. When the button is pressed, the callback applies the button's output handler to a `Click` (thus "sending" a `Click` to the button's output); the button's input handler ignores all input.

```
>  (button text) parent outputHandler =
>    createButton text parent          `thenIO` \ but ->
>    addButtonCallback but
>               (outputHandler Click)   `thenIO` \ _ ->
>    returnIO inputHandler
>   where
>    inputHandler a = returnIO ()
```

- The `label` fudget creates a label widget and returns an input handler which sets the label string to the input value's textual representation. The output handler is ignored since labels have no output.

```
>  (label text) parent outputHandler =
>    createLabel text parent           `thenIO` \ lab ->
>    returnIO (inputHandler lab)
>   where
>    (inputHandler lab) a =
>            setLabel lab (LabelString (show a))
```

- The `stateMachine` fudget creates a variable in which to store the state. The input handler returned applies the transition function `f` to the current state and the input value and then updates the state and applies the output handler to the output value.

```
>  (stateMachine f init) parent outputHandler =
>    newVar init                    `thenIO` \ stateVar ->
>    returnIO (inputHandler stateVar)
>   where
>    (inputHandler stateVar) a =
>      readVar stateVar               `thenIO` \ s ->
>      let (s', b) = f (s, a)
>      in  writeVar stateVar s'       `thenIO` \ _ ->
>          outputHandler b
```

- The fudget composition operation `<==<` creates two fudgets in order.

```
>  (f1 <==< f2) parent outputHandler =
>    f1 parent outputHandler  `thenIO` \ handler ->
>    f2 parent handler        `thenIO` \ inputHandler ->
>    returnIO inputHandler
```

The combinators described so far are, at most concerned with the *local* appearance of the interface: none are concerned with the overall layout of the widgets on the screen. This issue can be tackled in several ways.

1. The default layout of widgets on the screen is determined by the order in which the widgets are created: the first widgets created will be nearer the top or the left-hand side of their parent than the last widgets. Therefore, in a fudget of the form `f1 <==< f2`, (where data flows from right to left) `f1` will appear above or to the left of `f2`.

   A simple way of changing the layout of widgets is to change the order in which they are created. For example, using an alternative fudget combinator `>==>` :: `F` α β $\rightarrow$ `F` β γ $\rightarrow$ `F` α γ (in which data flows from left to right) it is possible to swap the order in which a fudget (or group of fudgets) is created.

   This combinator is a little tricky to implement because it is necessary to create the first fudget before its output handler is known. One way to implement this is to use a mutable variable as a "place-holder" until the second fudget is created (when the output handler for the first fudget will be known).[4]

2. Most widget sets provide an extensive range of layout modifiers which allow non-default layouts to be created. For example, the children of a Motif `RowColumn` widget will either be arranged in a row or in a column depending on the current value of the `XmNorientation` resource.

[4] Peyton-Jones has shown us a simpler solution based on the fixpoint monad operator `fixIO` :: (`IO` α) $\rightarrow$ `IO` α.

At present, we provide three such "fudget modifiers": `row, column, grid` (all of type `F` α β $\rightarrow$ `F` α β) which arrange the widgets in the fudgets they are applied to in a horizontal row, a vertical column or in a rectangular grid.

6 Problems with Fudgets

We have applied a set of Fudgets based on the above implementation technique to a large industrial project. Overall, our experience of using Fudgets is that they allow one to generate sophisticated interfaces quickly and easily. However, in some circumstances, we found the structured approach required when using Fudgets overly restrictive.

During the summer of 1993, the first author carried out a case study for BT — investigating the suitability of Functional Programming Languages for industrial use [6] — during which we implemented a front end for a small part of BT's database. A (simplified) version of a single screen had the following characteristics:

- When a button is pressed, a query (consisting of a name and an address) is to be sent to the database and the result displayed in an output field.
- Queries are to be "validated" before being sent to the database. (For example, one might check that the name is a non-empty sequence of letters.)
- In the event of an error (whether caused by failing the validation check or an unsuccessful query), an error shell must "popup" and display the error message.

Figure 4 shows the dataflow within this application.

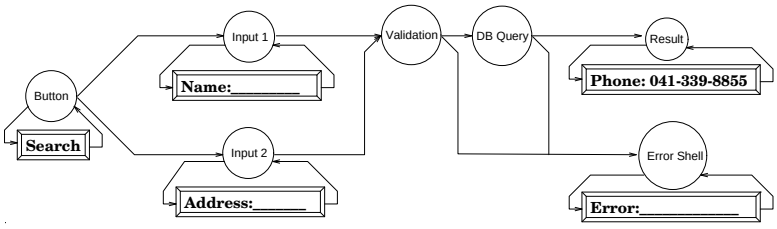

Figure 4. A Database Frontend

There are three major problems in turning this diagram into a valid fudget:

1. The two search strings must be received simultaneously by the validation section. With the above definition of Fudgets, we must send messages to the two input fudgets requesting them to output their current contents; there is no possibility of sending these messages "simultaneously" and so no possibility of receiving the responses "simultaneously."

   Our solution is to define a second kind of Fudget consisting of those widgets which cannot generate callbacks. Fudgets of this kind can be represented by the type

   `type F2` α β `= Widget` $\rightarrow$ `IO (`α $\rightarrow$ `IO` β`)`

   and combined with the combinator `>|< ::` `F2` $\alpha 1$ $\beta 1$ $\rightarrow$ `F2` $\alpha 2$ $\beta 2$ $\rightarrow$ `F2 (`$\alpha 1$`,` $\alpha 2$`) (`$\beta 1$`,` $\beta 2$`)` which, on receiving a pair of values, sends the first value to the first fudget and the second value to the second fudget and returns the two replies received.

```
>  (f1 >|< f2) parent =
>    f1 parent                              `thenIO` \ h1 ->
>    f2 parent                              `thenIO` \ h2 ->
>    returnIO (h1 `combine` h2)
>   where
>    (h1 `combine` h2) (a1, a2) =
>      h1 a1                                `thenIO` \ b1 ->
>      h2 a2                                `thenIO` \ b2 ->
>      returnIO (b1, b2)
```

This second kind of fudget is readily converted to the first kind. It is neither possible nor sensible to convert the first kind to the second kind.

Carlsson and Hallgren have confirmed [2] that their implementation of Fudgets suffers from a similar problem though they solve it in a different manner.

2. Both the validation section and the database lookup section have two outputs; with both Carlsson and Hallgren's Fudgets and our own, Fudgets are only allowed to have a single output.

   There are two solutions to this problem:

   - We might design the validation and database lookup fudgets so that they "tag" their output according to whether it is an error message or a valid result. Special "routing combinators" could use these tags to decide where to send the result.

     Problems with this solution are that they tend to introduce a lot of tagging and untagging functions into the program and also, because the dataflow is no longer explicit in the structure of the program, much of the benefit of the Fudget approach is lost.

   - An alternative solution is to provide a third kind of Fudget (and associated combinators) with a single input but two outputs. This is perfectly possible but we see no reason to suppose that we won't also find a need for still more kinds of Fudget with two inputs and two outputs, one input and three output, etc.

     The tension here is between providing a small but restrictive library of well-understood components or a larger library of less general, less well-understood components. We favour the former.

3. One aspect of the way the X window system organises windows in hierarchies is that popup shells must be children of the top-level shell to work correctly. On the other hand, we would probably want to use a fudget modifier such as `row` to control the layout of the other fudgets and so the other fudgets will not be parents of the top-level shell. With the widget combinators and modifiers discussed above, it is not possible for the input widgets and the error widget to have different parents.

   This is just a single instance of a basic problem: in widget-based programs, the overall structure of the visual layout of the interface is determined by the widget hierarchy; in fudget-based programs, the widget hierarchy is determined by the dataflow within the application. That is, the dataflow within the application will determine the overall structure of the visual layout of the interface. It follows that it may be hard or even impossible to obtain a particular visual layout without significantly sacrificing the clarity of the program.

It is worth noting that the last two problems are caused by the structured approach imposed by the use of combinators. In the first case, the problem is that a highly structured approach doesn't seem to be appropriate; while in the second case, there is a clash between the structure imposed by the dataflow and the structure imposed by the visual appearance required of the interface.

7 Conclusions

Till recently, there has been no easy way to create graphical user interfaces in lazy functional languages. Carlsson and Hallgren have implemented a complete widget (or, rather, fudget) set and combinators for combining simple fudgets to create complete applications. Though not without problems, their system is very impressive: it provides a fast and effective way of generating graphical user-interfaces.

The alternative fudget implementation described here suffers from many of the same problems as Carlsson and Hallgren's implementation but differs in one important respect: our implementation approach can be applied to standard widget sets. This has two advantages:

- Applications developed with our fudgets will be consistent with other applications developed with the same widget set (irrespective of which language they are implemented in). In particular, they will have the same "look and feel" and the same resource databases can be used to control their overall colouration, etc.
- It takes a substantial effort to create a widget set. By using pre-existing widget sets, we avoid the need to recreate that effort.

We see two ways of developing this work further:

- Many imperative programmers do not directly write programs like that shown at the end of section 2. Instead, they use a "GUI builder" which allows them to place widgets directly on the screen. The GUI builder automatically generates a program to which the programmer need only add the callback routines.

  We see this approach as a useful way of overcoming one of Fudgets main problems: obtaining the correct layout. We imagine that one would first write the application and then use the GUI builder to rearrange the widgets on screen.

  (We understand that there is a GUI builder for Carlsson and Hallgren's Fudget system though we have not been able to see this builder in operation.)
- Ideally, one would like to model each user interface component as a concurrent function (widget). User interface components then communicate with each other and the client program via streams of messages. This removes the need for callback functions, which are only needed in languages like C because they have no support for concurrency.

  Currently, the Glasgow Haskell compiler does not generate code which can be executed concurrently. However, this facility might be available in a future version. The authors have already built a concurrent X11 interface for Ada which results in much simpler software. An interface in a concurrent Haskell should benefit similarly.

Acknowledgements

The work reported here is based on an Openlook version developed while at Glasgow University's Computing Science Department. The Motif version and (a *considerably extended* variant of) the database example were developed by the first author while working at BT Research Labs on the FLARE project whose support we gratefully acknowledge. Thanks too to Will Partain and Simon Peyton-Jones for their patience in explaining how to extend the Glasgow Haskell Compiler.

References

[1] M. Carlsson and T. Hallgren. Fudgets: A Graphical User Interface in a Lazy Functional Language. In *Proceedings of the Conference on Functional Programming and Computer Architecture,* 1993.

[2] M. Carlsson and T. Hallgren. Private communication. 14 October, 1993.

[3] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional languages. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture,* pp. 636–666, Cambridge, Massachussets, USA, 26–28 August 1991.

[4] S.L. Peyton-Jones and P. Wadler. Imperative Functional Programming. In *Proceedings of the 1993 Conference on Principles of Programming Languages,* Charleston, ACM, 1993.

[5] J. Launchbury. Lazy Imperative Programming. In *Proceedings of the Workshop on State in Programming Languages,* pp. 46–56, Copenhagen, 1993. (Available as YALEU/DCS/RR-968, Yale University.)

[6] A. Reid. A Window-based Application Front-End in Haskell BT Research Labs, Martlesham Heath. September 1993.

[7] D. Sinclair. Lazy Wafe — Graphical Interfaces for Functional Languages. In Heldal et al., editor, *Glasgow Workshop on Functional Programming,* 1992.

[8] S. Singh. Using XView/X11 from Miranda. In Heldal et al., editor, *Glasgow Workshop on Functional Programming,* 1992.