Fudgets
A Graphical User Interface in a Lazy Functional Language

Magnus Carlsson, Thomas Hallgren
Chalmers University of Technology
{magnus,hallgren}@cs.chalmers.se

Abstract

This paper describes an implementation of a small window-based graphical user interface toolkit for X Windows written in the lazy functional language LML. By using this toolkit, a Haskell or LML programmer can create a user interface with menus, buttons and other graphical interface objects, without conforming to more or less imperative programming paradigms imposed if she were to use a traditional (imperative) toolkit. Instead, the power of the abstraction methods provided by Haskell or LML are used.

The main abstraction we use is the *fudget*. Fudgets are combined in a hierarchical structure, and they interact by message passing. The current implementation is based on a sequential evaluator, but by using non-determinism and oracles, we suggest how the fudgets can evaluate in parallel. We believe that the toolkit can be extended to a full-feathered and practically useful high level graphical toolkit.

1 Introduction

Not so long ago, the dominating way for a user to interact with a computer was by typing text on a keyboard and reading text of a screen. Today, this traditional text oriented user interface is being replaced by graphical user interfaces, where the user interacts with the computer by manipulating graphical objects on a screen with a pointing device, typically a mouse.

Graphic user interfaces are more flexible and therefore more complex to program. To deal with this extra complexity more levels of abstractions are used. As in all programming it is important to find the right abstractions. This has led to the development of Graphical User Interface (GUI) toolkits to simplify the application programmer's job.

A major advantage of functional programming languages over traditional imperative languages is the abstraction methods they provide: higher order functions, polymorphism and algebraic data types. This suggests that functional languages may be better equipped to handle the complexity of graphical user interfaces than traditional languages. But functional languages are often criticized for having poor I/O facilities, making it hard to write interactive programs, in particular programs with fancy graphical user interfaces.

The major goals with our work are to show:

- that the abstraction methods and I/O facilities provided by functional languages are adequate for implementing programs with graphical user interfaces, and
- that implementations of lazy functional languages are efficient enough to deal with the potentially large flow of data and swift responses required by a graphical user interface.

So, rather than defining an interface between an existing GUI toolkit (such as the Macintosh Toolbox or Motif) and a functional language, we choose to start from a lower level and implement a GUI toolkit in the functional language itself. This approach allows us to use the power of the abstraction methods provided by the functional language, instead of relying on abstractions designed for imperative languages. It also puts a larger part of burden of handling a GUI on the functional program, thus requiring the implementation to be more efficient to obtain good performance.

The functional languages we work with are Lazy ML [3] and Haskell [8] and the window system is X Windows [17]. The interface to X Windows goes through Xlib [11]. Except for one example in C, all code in the paper is given in Haskell.

The main abstraction we use is the *fudget*, the functional correspondence to what is called the widget in some traditional GUI toolkits. We have developed a library of fudgets implementing common user interface components, like buttons, menus, scroll bars, etc. Complex user interfaces are built up by combining fudgets in a hierachical structure, where the fudgets interact by message passing. There is no global state: state information, when needed, is encapsulated inside the fudgets, hidden from the outside world.

The remainder of this paper is organized as follows: we start with a brief introduction to the X Windows system and look at a small example program written in C using the Motif toolkit (Section 2). We then describe our approach to GUI program structuring in a lazy functional language and introduce the fudget type (Section 3) and present the same example now implemented using fudgets. In Section 4 we present a mechanism for automatic and dynamic layout of fudgets. Section 5 contains a larger fudget programming example. We present some details on how fudgets are represented (Section 6). With the chosen representation, we can easily add mechanisms for parallelism and nondeterminism, as is illustrated in Section 7. We take a quick look at the implementation of the interface to Xlib in Section 8. Related work is presented in Section 9 and conclusions are given in Section 10.

2 The X Windows system

In the X Windows system [17], you write a client program, which 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 client sends a stream of *commands* (for creating windows, drawing lines, writing text etc.) to the server and receives a stream of *events* (which tell the client about keystrokes, mouse button presses, motion of the mouse, etc.) from the server. Most 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 drawing is usually clipped by the window boundaries. This way, client programs only have to bother about their own windows and are usually completely unaware of the existence of windows controlled by other clients. Therefore, a user can handle many independent activities simultaneously, possibly on different computers in a network.

The windows have a 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 graphical 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 objects (like scroll bars, pulldown menus and buttons), often called *widgets* (*wi*ndow ga*dgets*), are built up by a number of windows in this way.

The root of this window tree is a window that simply covers the whole screen, and is usually filled with some background color or pattern. The children of the root window are usually so called *shell* windows. They have a title bar and it is usually possible to move them around and resize them by using the mouse. So the shell windows are the most "window like" windows, from the user's point of view.

In addition to the 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 client, telling it that the newly exposed part of the window needs to be redrawn.[1]

2.1 Imperative toolkit programming

Before introducing our functional toolkit solution, let us warm up by looking at a simple imperative program that uses a conventional toolkit. We will discover that program control is somewhat different from what we find in programs with simple text interfaces.

In traditional imperative X Window based toolkits, you create a tree of widgets and connect *callback* routines to them. They are called callback routines because there are usually no direct calls to them in your code, but the toolkit will call them in response to various events. We will name the code you write the *application*.

After creating the widget tree and specifying the callback routines, you enter the *main event loop*, where events are dispatched to the widgets, which in turn respond by calling the callback routines. In short, we could say that the toolkit converts *low level events*, such as "Mouse button is pressed at (x, y)", into *high level events*, such as calling the OK button callback routine. See Figure 1. The callback routines, in turn, can react with *high level commands*, such as "Pop up the Save dialog box"), by calling routines in the toolkit. The toolkit then emits a number of *low level commands* to carry out the high level command. Typically, there is also a number of low level events that the toolkit could handle more or less autonomously, such as expose events and requests for resizing windows. The toolkit somewhat resembles lower systems in the brain, controlling various functions of the body without bothering the cerebral cortex (the application code).

So the picture is that *the toolkit is in control*, handling the low level events and maintaining the visual state of the interface. Sometimes, application specific computation is necessary, and then the toolkit calls application code.

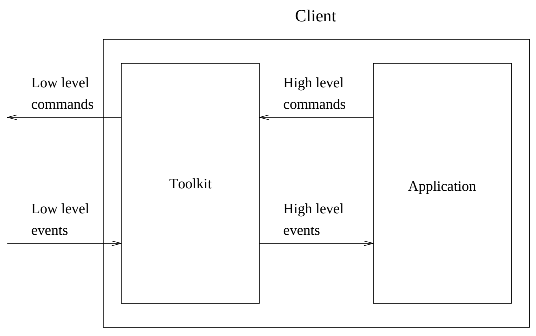


Figure 1: The structure of the client. The purpose of the toolkit is to take care of handling all low level commands and events. The toolkit can also emit high level events as a response on low level events. The high level events are handled by the application code, which in turn can emit high level commands.

2.2 The Motif counter example

Let us look at a small example with a window containing a button and a number display. Whenever the button is pressed, the number is increased by one. The example is written in C using the popular toolkit Motif [25]:[2]

```
static int count = 0;
static Widget display;

static void SetDisplay(Widget display, int i)
{
  char s[10];
  Arg wargs[1];

  sprintf(s, "%d", i);
```

[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.

[2] The example has been somewhat stripped; the callback arguments and arguments for determining various widget attributes are omitted, and so is the conversion between C-strings and Motif strings.

```
  XtSetArg(wargs[0], XmNlabelString, s);
  XtSetValues(display, wargs, 1);
}

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

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

  top = XtInitialize();
  row = XtCreateManagedWidget("row",
                        xmRowColumnWidgetClass, top);
  display = XtCreateManagedWidget("display",
                        xmLabelWidgetClass, row);
  button = XtCreateManagedWidget("button",
                        xmPushButtonWidgetClass, row);
  SetDisplay(display, count);
  XtAddCallback(button, (XtCallbackProc)increment,
                        (XtPointer)display);
  XtRealizeWidget(top);
  XtMainLoop();
}
```

The program starts with creating a shell widget called `top`, which will be the root of the widget tree. The rest of the tree is created with repeated calls of `XtCreateManagedWidget`, where the arguments specify what kind of widget to create, and where to put it in the tree. The widgets are:

- `row`, a layout widget which put all its children in a row or in a column.
- `display`, which shows a string which will be the count.
- `button`, a button that the user can press. Whenever this happens, an associated callback routine is called.

When the widget tree is created, the display is reset to show zero, and the C-function `increment` is registered as a callback routine for the button widget. `increment` increments the counter and updates the display widget.

3 Our approach

If we want to apply the callback style directly in a pure lazy functional toolkit, we must find out what it means to "call a routine". A straightforward solution would be to stick to the imperative style by using variations of the state monad [24]. This suggests a simple way of using an existing imperative toolkit in a functional program. It is likely though, that this will imply a imperative style throughout the program, so why then use a functional language at all?

Instead, we chose to use a stream processing style, with functions operating on streams of events and commands. As suggested by Figure 1, we can distinguish four types of streams, high level command and event streams, and low level ditos. Our toolkit consists of stream functions consuming high and low level events, and producing high and low level commands. They correspond to the widgets in traditional toolkits, and we call them *fudgets* (*Fu*nctional Wi*dgets*). When developing an application, you (the application programmer) write stream functions that handle high level messages and somehow connect them with the fudgets from the toolkit.

3.1 The fudget type

Let us take a closer look at the types of the four different streams. The low level command type has constructors corresponding closely to the drawing commands that you could send to the X server. Similarly, the low level event type mostly consists of constructors for the various events that the server could produce. These types are fixed and is something that the application programmer normally need not worry about.

The type of high level events and commands (which we will simply call *input* and *output*) cannot so easily be determined once and for all. For example, consider a fudget `textF` for displaying and entering a line of text. We want the input type to be `String`, telling that the fudget accepts new strings to show. Suppose we want the fudget to output the text value whenever the return key is pressed, this is indicated by having the output type `String` too. Similarly, imagine a push button fudget `buttonF` which could output a unit value (of type `()` in Haskell), whenever it is clicked, and could input a boolean value `True` or `False` to make it sensitive or insensitive to mouse-clicks.

It seems reasonable to have the type of the high level event and commands as *parameters* in the fudget type. We introduce the notation

$$\mathtt{F}\ \alpha\ \beta \tag{1}$$

for the type of fudgets with input type α and output type β. Thus, `textF` will have type `F String String`, and `buttonF` will have the type `F Bool ()`.

We will visualize the fudget as a circle with four pins, see Figure 2. The information flows through the fudget from right to left. The high level messages go through the upper pins, the low level events and commands through the lower pins. You can think of the lower pins as being connected directly to the fudget's window.

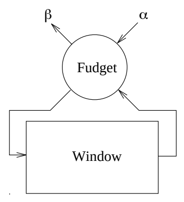


Figure 2: The fudget `F` α β.

3.2 Putting fudgets together

Complex graphic interfaces are constructed from simpler building blocks, so we need a set of combinators for this. A simple combinator would take two fudgets as arguments and put them "in parallel' into one composite fudget, and we will call this combinator `>+<`. It routes the low level commands and events to and from each fudget independently, so they exist side by side without having to bother about each other, each one controlling its own window. Since the composite fudget consists of two subfudgets, we need a mechanism for distinguishing the output from them, and adressing input to each one of them. For this reason, we introduce the type of disjoint union, called `Either` in Haskell:

```
data Either a b = Left a | Right b
```

Either α β will be abbreviated as $\alpha + \beta$. The type of >+< will then be

$$\texttt{F}\ \alpha_1\ \beta_1 \rightarrow \texttt{F}\ \alpha_2\ \beta_2 \rightarrow \texttt{F}\ (\alpha_1 + \alpha_2)\ (\beta_1 + \beta_2)$$

We use the constructors in Either to indicate that a high level message is sent to or from either the left or the right subfudget. Now, we can for example put together our text fudget and button fudget:

```
textF >+< buttonF ::  F (String + Bool) (String + ())
```

Say that we want to enable the button, this is done by sending Right True to the composed fudget.

3.2.1 Fudget composition

The pairing combinator allows us to put any number of fudgets together into one single fudget, but we need a means by which they can communicate high level information to one other. Normally in functional programming, this is done with an operator for *function composition* with type

$$(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

With this in mind, we introduce a combinator for fudget composition, which we will name >==<, with type

$$\texttt{F}\ \beta\ \gamma \rightarrow \texttt{F}\ \alpha\ \beta \rightarrow \texttt{F}\ \alpha\ \gamma$$

Just as >+<, >==< will put two fudgets together and let them control their windows independently, and in addition, the output from the right fudget is connected to the input of the left fudget. Consider the somewhat silly fudget

```
textF >==< textF
```

If text is entered in the right text fudget, it will be echoed in the left one. See also Figure 3.

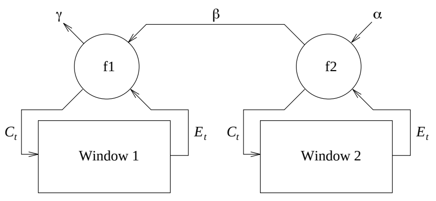


Figure 3: The fudget f1 >==< f2

3.2.2 Abstract fudgets

With the right set of primitive fudgets (such as textF and buttonF), we can now imagine rather complex interfaces being built. But we must also have the ability to combine this interface with the application specific code, corresponding to the right box in Figure 1. We will do this by an operator that lets the programmer turn an arbitrary stream function into a fudget. The operator is called absF and has type

$$\texttt{SP}\ \alpha\ \beta \rightarrow \texttt{F}\ \alpha\ \beta$$

(SP is an abbreviation for Stream Processor). We also need to tools for writing stream functions. This is done in a continuation style with the functions

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

getSP (\a -> sf) is a stream function that will wait for a message and then become the stream function sf. putSP l sp will output all the messages in the list l and then become the stream function sf.

A useful function derived from these is mapSP (cf. the standard map function on lists) of type

$$(\alpha \rightarrow \beta) \rightarrow \texttt{SP}\ \alpha\ \beta$$

3.2.3 The counter example with fudgets

We can now construct a fudget with the same behavior as the simple counter program in Section 2.2. The fudget consists of a button, an abstract fudget that does the counting, and an integer display fudget intDispF of type F Int α[3] (See Figure 4).

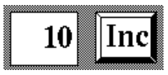


Figure 4: A small counter

We use fudget composition to connect the button with the counter and the counter with the display:

```
intDispF >==< (absF counter) >==< buttonF
```

Here, counter has type SP () Int and can be defined as[4]

```
counter = count 0
          where count n = getSP $ \ _ ->
                          putSP [n+1] $
                          count (n+1)
```

The three fudgets can be seen in Figure 5.

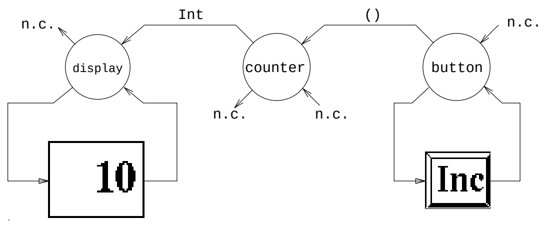


Figure 5: Whenever the button fudget is pressed, it sends a () to counter, which counts the number of clicks and sends this count to the display. Connectors marked n.c. are not connected.

3.2.4 Loops

If fudgets need to exchange information both back and forth, they can be connected with the operator loopLeft of type

$$\texttt{F}\ (\alpha + \beta)\ (\alpha + \gamma) \rightarrow \texttt{F}\ \beta\ \gamma$$

Whenever the enclosed fudget outputs an α message, it is fed back into the fudget.

As an example, consider the definitions

[3]The type variable on the output indicates that the fudget never outputs anything.
[4]$ is function application in Haskell

```
stripEither :: Either a a -> a
stripEither Left  a = a
          | Right a = a

loopAll :: F a b -> F c d
loopAll f = loopLeft (absF (mapSP Left) >==< f >==<
                      (absF (mapSP StripEither)))
```

Now, with

```
loopAll (textF >==< textF)
```

we get two text fudgets, and if text is entered in one of them, it is echoed in the other[5] (See also Section 3.2.1).

3.2.5 Some derived combinators

The following combinators and operators are not necessary since they can be derived from the ones introduces sofar, but they are quite useful and some of them are actually implemented more efficiently than the following definitions suggest:

```
(>^^=<) :: SP b c -> F a b -> F a c
p >^^=< f = absF p >==< f

(>=^^<) :: F b c -> SP a b -> F a c
f >=^^< p = f >==< absF p

(>^=<) :: (b -> c) -> F a b -> F a c
p >^=< f = mapSP p >^^=< f

(>=^<) :: F b c -> (a -> b) -> F a c
f >=^< p = f >=^^< mapSP p
```

With these, `loopAll` in the previous section could be written

```
loopAll f = loopLeft (Left >^=< f >=^< stripEither)
```

3.2.6 The list combinator

Sometimes you want to create a list of similar interface objects (examples are button panels, menu choices or file lists). For this purpose, we introduce the list fudget combinator `listF` :: $[(\tau, \mathrm{F}\ \alpha\ \beta)] \to \mathrm{F}\ (\tau,\alpha)\ (\tau,\beta)$. It combines a list of tagged fudgets of some type into one fudget, where the high level in- and outgoing messages are tagged to determine destination or source, respectively.

3.3 A Haskell program with fudgets

We have learned how to compose a fudget from the primitive fudgets and application specific abstract fudgets. We will now put this fudget into a shell fudget (corresponding to the `top` shell widget in the Motif example) and present a Haskell program with this fudget.

The shell fudget wraps a shell window with a title bar around an enclosed fudget, and it is called `shellF` :: `String` $\to \mathrm{F}\ \alpha\ \beta \to \mathrm{F}\ \alpha\ \beta$. As the type suggests, the high level messages are simply passed through the shell fudget.

Before presenting the Haskell program, we will briefly describe how input/output is done in Haskell. The input/output model is stream based, and a Haskell program must contain a function `main` of type `Dialogue`, where

[5]For this to work, `textF` must only output the content when it is altered by the user, not when a new content is input from the other fudget. Otherwise, we get an infinite loop.

```
type Dialogue = [Response] ->[Request]
```

A Haskell program is a stream function that outputs *requests* (such as "Write this string to that file") to the outer world, and consumes *responses* (i.e. "Ok" or "That file is write protected!") from it.[6]

So we introduce a function `fudlogue` (*fud*get dia*logue*), with type $\mathrm{F}\ \alpha\ \beta \to$ `Dialogue`. `fudlogue` will turn the low level commands from the fudget into suitable requests, and extract low level events from the response stream to feed back into the fudget.

Now, let us look at the counter example as a Haskell program:

```
module Main(main) where
import Fudgets

main      = fudlogue (shellF "Counter" counter_f)

startstate = 0

counter_f  = intDispF startstate  >==<  absF counter
             >==<  buttonF "Inc"

counter    = count startstate
             where count n = getSP $ \ _ ->
                             putSP [n+1] $
                             count (n+1)
```

Here, we use more practical versions of the `intDispF` and `buttonF` fudgets, with additional parameters for the initial state and the button name, respectively.

4 Dynamic layout

The simplest approach to layout (from the toolkit implementor's point of view) is to let each fudget take an extra argument defining the window geometry. (The geometry determines the height and width of a window, and where it is placed in its parent window.) The programmer can then completely control where to put the fudgets. This is not the level one usually wants to work on, however.[7]

We implemented a simple layout scheme to make application programming easier, by inventing a layout-conscious cousin of `>+<`, called `>+#<`:

```
>+#< :: F a b -> (Distance, Orientation, F c d)
             -> F (a + b) (c + d)
```

Here, `Distance` is the distance between the fudgets in pixels, and the `Orientation` argument specifies how the first fudget should be placed relative to the second:

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

The composition combinator `>==#<` is defined similarly.

4.1 Drawbacks with this layout mechanism

The layout mechanism described has two obvious drawbacks. Firstly, the layout of fudgets is connected to the structure of the program. There is no easy way of saying that two fudgets should be placed together if they are not

[6]There are other means of doing input/output in Haskell, see [8].

[7]Parts of the problem could be solved by using a graphical layout program which lets you place and resize fudgets with the mouse. Code with explicit window geometry information will be generated by the layout program. A prototype layout program has been developed for the FUDGETS library [1].

combined in the program. Secondly, the program is somewhat cluttered with a lot of layout arguments which possibly hide the fudget structure.

A solution is to wrap a layout filter around the combined fudgets, where the programmer specifies to the layout filter how the subfudgets should be placed. This allows a more "global" placement.

5 A larger example: Life

Let us look at a somewhat more complicated example, a simulator for Conway's game of Life. The user will see two shell windows, a board window with cells and a button panel, controlled by the fudgets `board` and `panel` (See Figure 6). The user can click anywhere on the board to insert or remove cells or resize the board. The size of the cells is chosen from a radio group in the panel, which also has a toggle button for starting and stopping the simulation, and a quit button. In the following sections, we will take a closer look at `board` and `panel`.

When the simulation is started, new generations are computed and shown at regular intervals. A generation can be represented as a list of the cells which are alive, together with the bounds of the board:

```
type Cell = Bool
type Pos = (Int,Int)
type Bounds = Pos
type Generation = [Pos]
```

5.1 The Life fudget

The fudget `board` should visualize a generation. It should be easy to update either the whole generation or any individual cell in the shown generation. It must also permit the cell size to be changed. We capture this with the types

```
type CellSize = Int
data LifeCmds = NewCell (Pos, Cell) | NewGen Generation
              | NewCellSize CellSize
```

We want `board` to report when the user clicks at a certain position with the mouse, and when the window is resized. So the high level events are defined as

```
data LifeEvts = MouseClick Pos | NewBounds Bounds
```

and `board` will get the type `F LifeCmds LifeEvts`.

5.2 The Panel fudget

The type of `panel` will be `F (Bool` + `CellSize) (()` + `CellSize)`. When the toggle button is on, the panel will output a `()` as a *tick* at regular intervals[8], and if the user chooses a new cell size, the size is output. The input type of `panel` indicates that it can be used to control the settings of the toggle and the radio group, but we will not use this possibility.

5.3 The control stream function

The stream function `control` will receive the ticks from `panel`. On each tick, it will compute a new generation and output this. Clearly, the output from `control` must be connected to the input of `board`. But `control` must know when the user has clicked in the board window, so the output from `board` must go to `control` as well. We need to use the loop combinator to connect them (See also Figure 6):

[8]This is implemented by having the runtime system sending special timer alarm events to the Haskell program

```
control :: SP (LifeEvts + (() + CellSize)) LifeCmds
toplevel = loopLeft ((Left >^=< board) >=^^< control)
           >==< panel
```

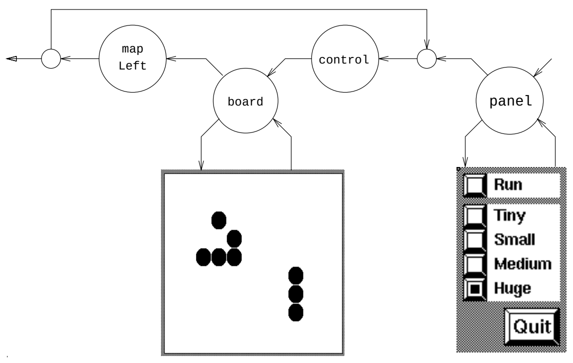


Figure 6:

We will not go into more detail about the internal structure of `board`, `control` or `panel`, but simply leave this example now.

5.4 Dynamic creation of fudgets

Our examples so far have had a static fudget structure, as indicated by the figures. Now, we will introduce a higher order combinator that can be used to create fudgets on the fly: `dynListF` :: $\mathtt{F}\ (\tau, (\mathtt{F}\ \alpha\ \beta) + \alpha)\ (\tau, \beta)$. A message to `dynListF` is tagged and can be either a new fudget $\mathtt{F}\ \alpha\ \beta$ or a message β to an already created fudget. (Cf. the description of `listF`.) The tag is used to associate fudgets with their messages.

Suppose we have a fudget `noteF` which is a small notepad. Now, we want to have a control panel that will let us create notepads and other handy little tools at will. Since we might want an arbitrary number of notepads, they have to be created dynamically. A simple control panel with just one button for creating notepads would be

```
module Main(main) where
import Fudgets
import NoteF(noteF)

main = fudlogue (shellF "Applications" newNoteFButton)
newFudget :: F (F a b) (Int,b)
newFudget = dynListF >=^^< countSP 1

newNoteFButton = newFudget >==<
    (const noteF >^=< buttonF "New Notepad")

countSP :: Int -> SP a (Int,a)
countSP n =
    getSP $ \x->
    putSP [(n,x)] $
    countSP (n+1)
```

6 Representation of Fudgets

Writing applications using the FUDGETS library will mostly consist of combining existing fudgets and writing abstract fudgets. So, most of the time you need not worry about the details of how fudgets are represented. But when implementing new primitive fudgets, you need to know some of the details.

As we have seen (c.f. Section 3.1), fudgets are stream processors with two input streams and two output streams. In the current implementation fudgets are represented as

```
type F a b = SP (TEvent + a) (TCommand + b)
```

where `TEvent` and `TCommand` are the types representing low level events and commands. The high and low level streams are merged into single streams allowing us to use the stream processor type also for fudgets. Fudget combinators, like `>+<` and `>==<`, take care of the details of separately routing low and high level messages to the appropriate places.

6.1 Representation of Stream Processors

Streams processors can be represented in several ways. In a lazy language, streams can naturally be represented as lists:

```
type SP a b = [a] -> [[b]]
```

This is the definition used in the current implementation of the FUDGETS library. Notice, though, that the definition is internal to the FUDGETS library and not visible to application programmers. Stream processors are constructed using operations like `putSP` and `getSP`.

We also need an operation to compose stream processors in parallel:

```
parSP ::  SP a1 b1 -> SP a2 b2 -> SP (a1+a2) (b1+b2)
```

To make it possible to deterministically merge the output streams from two stream processors composed in parallel we impose the following restriction on all stream processors `sp`:

$$\forall n.\mathtt{sp}\ (i_1 : i_2 : \ldots : i_n : \bot) = o_1 : o_2 : \ldots : o_n : o \qquad (2)$$

This means that there is a one-to-one correspondence between elements in the input output lists. This allows the order between the elements in the output stream of a parallel composition `parSP sp1 sp2` to be determined from the input stream. For example, if `Left x` appears at some point in the input stream, the next element in the output should be `Left y`, where `y` is the next element in the output stream from `sp1`.

The above restriction is also the reason why the output stream is represented as a list of lists rather than just list.

A problem with this representation of stream processors is that a straightforward implementation of `parSP`, causes a nasty space leak. `parSP` can be defined something like

```
parSP sp1 sp2 is = merge is (sp1 (onlyLeft is))
                            (sp2 (onlyRight is))
```

The problem is that there are two references to the input stream `is`. Now, if a long initial segment of the input stream to `parSP sp1 sp2` contains only elements for `sp1`, merge will take elements only from the output of `sp1` and thus leave the subexpression `(sp2 (onlyRight is))` unevaluated with its reference to the beginning of the input stream. This is really annoying, because we know that the large fragment kept in memory for `is` really is garbage, since it will be thrown away by `onlyRight` as soon as we try to evaluate it!

In the early days, the FUDGETS library suffered severely from this space leak. A surprisingly simple method to eliminate space leaks of this kind [20], has been successfully applied to the FUDGETS library.

7 Parallelism and nondeterminism

Maintaining a graphical user interface is really a task that is parallel in its nature, if you regard it as simultaneously view and update different parts of the interface. This is something that we would like to capture by permitting stream processors to evaluate in parallel, merging their output streams nondeterministically. One fudget could then be busy updating a complicated drawing for example, while other fudgets could respond to user actions. We will now introduce the `choose` operator, which makes this possible.

7.1 Parallel evaluation with `choose` and oracles

The operator `choose` has been implemented for doing nondeterministic programming in LML [2]. It has the type

```
choose:  Oracle -> a -> b -> Bool
```

`choose o a b` will evaluate the arguments `a` and `b` to WHNF in parallel (possibly using time slicing), and return `True` if `a` terminates first, otherwise `False`. The oracle `o` is consulted to determine this, and if the same oracle is used once again in another `choose` expression, that will immediately evaluate to the same boolean value. Hence, referential transparency is preserved:

```
f (choose o a b) (choose o a b)
```

is equivalent to

```
let b = choose o a b in f b b
```

Obviously, `choose` is not very useful when applied to only one oracle in a program. You need an everlasting supply of oracles. This is provided by the value `oracletree :: OracleTree` where

```
data OracleTree = MkOnode OracleTree Oracle OracleTree
```

A complication is that you have to distribute this oracle tree over those parts of your program that need nondeterministic choice. At first, it seems like we are forced to add an extra oracle tree argument to all our fudgets, whether or not they will use it. But there is a better solution, and that is to send the oracle tree as the first element in the event stream. The `>+<` and `listF` combinators take care of splitting the tree, so that each fudget will get its own subtree. This way, deterministic fudgets need not know at all about the oracles.

Our streams can now be represented as lists, and to merge two streams, we can use `pmergeEither`:[9]

```
pmerge :: [Oracle] -> [a] -> [a] -> [a]
pmerge (o:os) as bs =
    let (e:es,unes) = if choose o as bs
                      then (as,bs) else (bs,as)
    in e : merge os es unes

pmergeEither :: [Oracle] -> [a] -> [b] -> [a + b]
pmergeEither os as bs =
    pmerge os (map Left as) (map Right bs)
```

7.2 A more general fudget type

Consider a more general fudget type $\mathrm{F}_\Omega\ \alpha\ \beta = [E] \rightarrow \alpha \rightarrow ([C], \beta)$ where typically $\alpha = [\alpha_1]$, $\beta = [\beta_1]$. With this type, the loop combinator from Section 3.2.4 is

[9]The definition of `pmerge` is from [7], where it is used in implementations of real-time multi-user games in LML.

not needed as a primitive to recursively connect the different high level streams of our fudgets, instead we name the streams directly. The pairing combinator >+< will have the type $F_\Omega\ \alpha_1\ \beta_1 \rightarrow F_\Omega\ \alpha_2\ \beta_2 \rightarrow F_\Omega\ (\alpha_1, \alpha_2)\ (\beta_1, \beta_2)$. Here is how we could define the loop combinator:

```
loopLeft :: F (a,b) (a,c)
loopLeft f evs inp =
  let (cmds, (l,out)) = f evs (l,inp)
  in (cmds, out)
```

The function pmergeEither and the oracles introduced in the previous section can then be used to merge the high and low level event streams if that is needed inside fudgets.

More issues about fudgets and parallel evaluation can be found in [4].

8 Implementation

The FUDGETS library is built on top of Xlib [11], which contains a number of routines for creating and managing windows, rendering, reading events, etc. So, the implementation consists of two parts: the FUDGETS library itself and an interface to Xlib.

The implementation (source and documentation) is available via anonymous ftp [5]. The FUDGETS library is written in LML and consists of about 4000 lines of code. The Xlib interface is outlined below.

8.1 Implementation of the interface to Xlib

The facilities provided by XLib have been made available to the functional programs by extending the Haskell I/O system [8] (which can be used also in LML programs) with a few new requests and responses:

```
data  Request =
     -- file system requests:
             | ReadFile      String
             | WriteFile     String String
             .
             .
     -- New requests for Xlib interface
             | XDoCommand    XWId XCommand
             | XGetEvents

data  Response = Success
               | Str String
               | StrList [String]
               .
               .
     -- New responses for Xlib interface
             | XEventList [XEvent]
```

The type XCommand contains constructors corresponding to routines in Xlib. The type XEvent correspond to the type XEvent defined in Xlib. About 40 commands and 40 event types are currently supported. Apart from these two types, a number auxiliary types used in Xlib have been given analogous definitions in Haskell/LML.

Thanks to the integration of the XLib interface with the Haskell I/O system, fudgets can output not only X commands, but any I/O request, and receive responses. Thus, ordinary I/O operations can be performed inside fudgets.

A few lines of C code for every Xlib call and other constructor, have been added to the run-time system to implement the interface. Using the C monad [9] (not currently supported by the Chalmers Haskell compiler), most of this can be written directly in Haskell instead.

9 Related work

To our knowledge, FUDGETSis the first implementation of a toolkit in a lazy functional language that is not built on top of an existing toolkit.

A number of interfaces for functional languages have been built on top of existing toolkits, for example Lazy Wafe by Sinclair [18], XView/Miranda by Singh [19] and MIRAX by Tebbs [21]. In general, these interfaces lack combinators useful for structuring large applications.

9.1 eXene

eXene, by Reppy and Gansner [14, 6], is a toolkit for X Windows and Standard ML of New Jersey. It is written on top of Concurrent ML (CML) [13], and is thus multi-threaded. eXene aims towards being a full-fledged toolkit, completely written in SML (including the communication with the X server).

Events from the X server and control messages between parents and children are distributed in streams (coded as CML event values) through the window hierarchy, where each window has at least one thread taking care of the events. Drawing is done in a imperative style, by calling drawing procedures. High level events are reported either imperatively or by message passing: e.g., when a button is pressed, a callback routine is called, or a message is output on a channel.

9.2 Interactions

In [22], Thompson uses *interactions* to do I/O:

```
type Interaction a b = (Input,a) -> (Input,b,Output)
```

An Interaction α β is a function that, when applied to the input stream, will consume some input and return the rest, together with some output commands. It also transforms some value α into a β value.[10] Interactions can be composed by the sequential composition operator

```
sq :: Interaction a b -> Interaction b c -> Interaction a c
sq i1 i2 (in,st) = (rest,st2,out1++out2)
    where (in1,st1,out1) = i1 (in,st)
          (in2,st2,out2) = i2 (in1,st1)
```

These interactions have been used by Tebbs to implement an X Window interface in Miranda on top of an imperative toolkit written in C [21].

Having polymorphic input and output, the interactions resemble our fudgets. The difference is that all interactions are serially connected, where each interaction consumes a bit of event stream (input) and prepends a bit of command stream (output), whereas the fudgets are organized in a tree with the event stream being split and distributed over it, resulting in a number of fudget command streams being collected in one single stream.

Whereas the interactions and dialogues might be good for text-based I/O, we do not find them appropriate for dealing with the parallel nature of a GUI.

9.3 Concurrent Clean input/output

Concurrent Clean is a lazy language, where parts of the program can be evaluated in parallel [10]. The type system is

[10]The interactions are a generalization of Dwelly's *Dialogue combinators*, which have the same type on the input and output values.

extended with so called *unique types*, which very much resemble linear types. In [12], objects of unique types are used to model different aspects of the operating system, and functions for manipulating these objects can have instant real world effects, since the objects are unshared. This opens the possibility to do I/O 'inside' the program. A graphical user interface system has been implemented on top of the Macintosh toolbox as well as the Open Look toolkit for X Windows. The connection to these toolkits gives the programs an imperative touch, where you have a user defined program state which is manipulated by action functions triggered by the user choosing menu commands, for example.

10 Conclusions

We have implemented a subset of a GUI toolkit, the FUDGETS library, which can be extended to a full-feathered and practically useful high level graphical interface toolkit.

With a small reservation concerning efficiency, we believe that the goals stated in the introduction are met. The fudget concept has proved to be a useful structuring tool when developing programs with GUIs, allowing large programs to be built in a structured way. As a spinoff, the fudget concept has also been used to do standard Haskell I/O. The fudgets can emit any kind of request, and the response will be routed back to the fudget.

It should be noted, that this is still work in progress. We lack the experience of writing a really large application using the FUDGETS library.

The efficiency is in most cases adequate. Our test applications start up in a few seconds (running on a SPARCstation IPX). Response times are short. The rendering in response to e.g. the user pressing a button or selecting a menu item is in general as immediate as in conventional X programs. Some operations are slower, e.g. adjusting the sizes and positions of all the buttons in the calculator when the user resizes the window. Sometimes, you notice "the embarrassing pause" caused by garbage collection.

The garbage collection (GC) pause in our test application is in most cases less than 0.2s. The GC time is proportional to the size of live data with the copying garbage collector we currently use, so applications dealing with large data structures may suffer more from the GC pauses. We hope, however, that program transformations that reduce the amount of garbage generated and/or an appropriately tuned generational garbage collector [16] can be used to solve this problem.

10.1 Sample applications

We have implemented a number of small applications using the FUDGETS library: `calc`: a pocket calculator providing infinite precision rational numbers; `clock`: a transparent clock; `graph`: a program for viewing graphs of real valued functions of one real variable. The program allows the user to zoom in/out, differentiate functions, and search for roots; `life`: an implementation of Conway's game of life. (See Section 5 for a more detailed description); `sss`: a simple spread sheet; `xlmls`: a GUI to a previously written program to search for functions in the LML library by type [15]; `xmail`: a simple mail reader; `guit`: a graphical user interface builder for the FUDGETS library.

Figure 7 shows a screen dump with most of these programs, and some more.

10.2 Future work

By means of parallel evaluation and oracles, it seems like we could come even closer of capturing the parallel nature of a GUI, and permit a more natural way of connecting stream functions. Therefore, a tempting experiment would be to modify FUDGETS in this direction.

A source of inefficiency in many functional programs is the repeated destruction and reconstruction of data structures. The event and command streams processed by fudgets is a typical example of this. It would be interesting to see to what extent automatic program transformation, like deforestation [23], can be used to eliminate this inefficiency.

11 Acknowledgements

We wish to thank Lennart Augustsson, for his assistance in the extension of the run-time system to support the Xlib interface. Jan Sparud's fix of the space leaks suddenly made our programs much more useful. Jan, Lennart and Niklas Röjemo also did proof-reading and suggested various improvements.

John Launchbury pointed out that the C-monads could be used to implement the calls to Xlib.

References

[1] C. Ahlberg. GUIT, a Graphical User Interface Builder for the FUDGETS Library. In *Proceedings of the Winter Meeting*. Department of Computer Sciences, Chalmers, January 1993.

[2] L. Augustsson. Non-deterministic Programming in a Deterministic Functional Language. PMG Memo 66, Department of Computer Sciences, Chalmers University of Technology, S–412 96 Göteborg, 1988.

[3] L. Augustsson and T. Johnsson. *Lazy ML User's Manual*. Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the LML compiler.

[4] M. Carlsson. Fudgets - Graphical User Interfaces and I/O in Lazy Functional Languages. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, 1993.

[5] M. Carlsson and T. Hallgren. The FUDGETS library. Chalmers University. Anon. FTP: `ftp.cs.chalmers.se:/pub/haskell/chalmers/lml-<version>.lmlx.tar.Z`, March 1993.

[6] E.R. Gansner and J. Reppy. The eXene widgets manual. Cornell University. Anon. FTP: `ramses.cs.cornell.edu:/pub/eXene-doc.tar.Z`, June 1991.

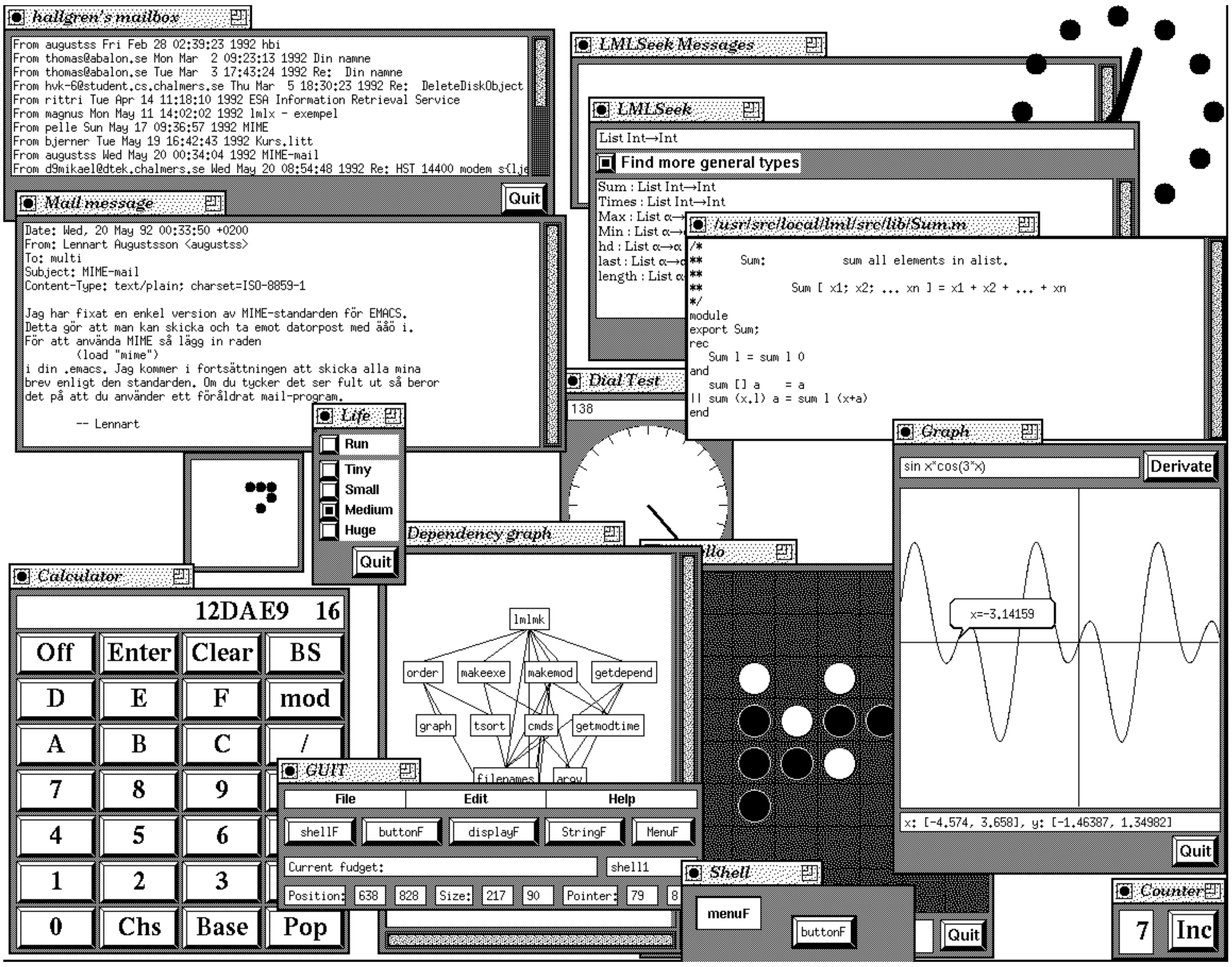
[7] T. Hallgren. Introduction to Real-time Multi-user Games Programming in LML. Technical Report Memo 89, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, January 1990.

[8] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[9] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 1993 Conference on Principles of Programming Languages*, 1993.

[10] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeyer. Concurrent clean. In *Proceedings of the PARLE'91 Parallel Architectures and Languages Europe conference (LNCS 505)*, Eindhoven, June 1991.

[11] A. Nye. *Xlib reference manual, volume 2*. O'Reilly & Associates, Inc., 1990.



```
From augustss Fri Feb 28 02:39:23 1992 hbi
From thomas@abalon.se Mon Mar  2 09:23:13 1992 Din namne
From thomas@abalon.se Tue Mar  3 17:43:24 1992 Re:  Din namne
From hvk-6@student.cs.chalmers.se Thu Mar  5 18:30:23 1992 Re:  DeleteDiskObject
From rittri Tue Apr 14 11:18:10 1992 ESA Information Retrieval Service
From magnus Mon May 11 14:02:02 1992 lmlx - exempel
From pelle Sun May 17 09:36:57 1992 MIME
From bjerner Tue May 19 16:42:43 1992 Kurs.litt
From augustss Wed May 20 00:34:04 1992 MIME-mail
From d9mikael@dtek.chalmers.se Wed May 20 08:54:48 1992 Re: HST 14400 modem s{lj
```

```
Date: Wed, 20 May 92 00:33:50 +0200
From: Lennart Augustsson <augustss>
To: multi
Subject: MIME-mail
Content-Type: text/plain; charset=ISO-8859-1

Jag har fixat en enkel version av MIME-standarden för EMACS.
Detta gör att man kan skicka och ta emot datorpost med åäö i.
För att använda MIME så lägg in raden
        (load "mime")
i din .emacs. Jag kommer i fortsättningen att skicka alla mina
brev enligt den standarden. Om du tycker det ser fult ut så beror
det på att du använder ett föråldrat mail-program.

        -- Lennart
```

```
List Int→Int
Sum : List Int→Int
Times : List Int→Int
Max : List α→
Min : List α→
hd : List α→α
last : List α→
length : List α
```

```
/*
**      Sum:            sum all elements in alist.
**
**              Sum [ x1; x2; ... xn ] = x1 + x2 + ... + xn
*/
module
export Sum;
rec
   Sum l = sum l 0
and
   sum [] a    = a
|| sum (x.l) a = sum l (x+a)
end
```

Figure 7: A collage of fudget applications. All windows belong to programs developed with the FUDGETS library.

[12] J. von Groningen P. Achten and R. Plasmeijer. High Level Specification of I/O in Functional Languages. In *Proc. of the International Workshop on Functional Languages*. Springer Lecture Notes in Computer Science, 1992. Anon. FTP: ftp.cs.kun.nl:/pub/Clean/papers/CleanIOPaper.ps.Z.

[13] J. Reppy. CML: A Higher-order Concurrent Language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[14] J. Reppy and E.R. Gansner. The eXene library manual. Cornell University. Anon. FTP: ramses.cs.cornell.edu:/pub/eXene-doc.tar.Z, June 1991.

[15] M. Rittri. Using types as search keys in function libraries. *J. of Functional Programming*, 1(1):71–89, 1991. Earlier version in *Func. Prog. Lang. and Comput. Arch.*, 4th Conf., ACM Press 1989.

[16] N. Röjemo. Generational garbage collection is Leak-Prone. Submitted to FPCA 1993, December 1992.

[17] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2), April 1986.

[18] D.C. Sinclair. Lazy Wafe - Graphical Interfaces for Functional Languages. Departement of Computing Science, University of Glasgow, 1992. Draft.

[19] S. Singh. Using XView/X11 from Miranda. In Heldal et al., editor, *Glasgow Workshop on Functional Programming*, 1991.

[20] J. Sparud. Fixing Some Space Leaks without a Garbage Collector. Submitted to FPCA 1993, December 1992.

[21] M. Tebbs. MIRAX - An X-window Interface for the Functional Programming Language Miranda. Technical report, School of Engineering and Applied Science, April 1991.

[22] S. Thompson. Interactive Functional Programming. In D.A. Turner, editor, *Research topics in Functional Programming*. Addison-Wesley Publishing Company, 1990.

[23] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358, Nancy, March 1988.

[24] P. Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.

[25] D.A. Young. *The X window System : programming and Applications with Xt. OSF/Motif Edition*. Prentice Hall, 1990.