The Yale Haskell X Window Interface

Sheng Liang and John Peterson

Research Report YALEU/DCS/RR-972

Yale University

Department of Computer Science

New Haven, CT 06520

1 Introduction

The Yale Haskell X interface is built on top of the Common Lisp X Interface (CLX). Readers should refer to the CLX manual for a complete description of Xlib functions. This document contains a quick tour of Haskell X interface. Go through it before attempting any X Window programming in Haskell.

1.1 The I/O System

Yale Haskell builds its I/O system using a *monad*. The I/O monad uses a special data constructor, `IO`, in the result type of functions which involve the global state. Most X window functions have global effects and are only callable from the monad. The monad is described elsewhere.

1.2 Haskell X Interface and CLX

Most Haskell X functions have a CLX counterpart. We use a simple name mapping scheme. For example, Haskell function **`xWindowEventMask`** corresponds to the CLX function **`xlib:window-event-mask`**.

Some CLX objects are settable. In CLX, we can say:

```
(setf (xlib:window-event-mask window) mask)
```

In Haskell, we accomplish the above using a separate function:

```
xSetWindowEventMask window mask
```

The Haskell X interface tries to retain function arguments and their ordering as those in CLX. One exception is that arguments related to geometry are abstracted into the following Haskell data types:

```
data XPoint = XPoint Int Int                       -- x, y
data XSize  = XSize  Int Int                       -- width, height
data XRect  = XRect  Int Int Int Int               -- x, y, width, height
data XArc   = XArc   Int Int Int Int Float Float -- x, y, width, height, angle1, angle2
```

Many CLX functions return either an object or "null". We introduce a Haskell data type **`XMaybe`** for this purpose.

```
data XMaybe a = XSome a
              | XNull
```

For example, CLX function **`xlib:window-colormap`** returns a window's colormap or **`null`** in case the window does not have one. The corresponding Haskell function is:

```
xWindowColormap :: XWindow -> IO (XMaybe XColormap)
```

We also use **`XMaybe`** to handle optional function arguments.

1.3 Error handling

There is no explicit flow of control in a purely functional language. Indeed Haskell does not provide catch-and-throw style error handling (such as the exception mechanism in SML). However, the IO monad gives us the needed sequencing to capture and handle IO errors.

```
data XError  =  XError String
xHandleError :: (XError -> IO a) -> IO a -> IO a
```

For example, the following code passes the error message to `err_cont` when something goes wrong in any of the IO actions.

```
doIO err_cont =
  xHandleError (\ (XError msg) -> err_cont msg) $
  ... IO action 1 ... `thenIO` \ res1 ->
  ... IO action 2 ... `thenIO` \ res2 ->
  ... other IO actions ...
```

2 A Simple Example

The following simple window program functions as a "white board". It opens up a window, and allows the user to draw lines by moving the mouse while pressing the mouse button. This program is supplied in the demo directory (`$HASKELL/progs/demo/X11/draw/draw.hs`). It must be interrupted when you wish to exit it.

2.1 Code Listing

```
module Draw where

import Xlib

main = getEnv "DISPLAY" exit (\ host -> draw host)

draw :: String -> Dialogue
draw host =
  xOpenDisplay host `thenIO` \ display ->
  let (screen:_) = xDisplayRoots display
      fg_color = xScreenBlackPixel screen
      bg_color = xScreenWhitePixel screen
      root = xScreenRoot screen
  in
  xCreateWindow root
                (XRect 100 100 400 400)
                [XWinBackground bg_color,
                 XWinEventMask (XEventMask [XButtonMotion, XButtonPress])]
  `thenIO` \window ->
  xMapWindow window `thenIO_`
  xCreateGcontext (XDrawWindow root)
```

```
                [XGCBackground bg_color,
                 XGCForeground fg_color] `thenIO` \ gcontext ->
  let
    handleEvent :: XPoint -> Dialogue
    handleEvent last =
      xGetEvent display `thenIO` \event ->
        let pos = xEventPos event
        in
        case (xEventType event) of
          XButtonPressEvent  -> handleEvent pos
          XMotionNotifyEvent ->
            xDrawLine (XDrawWindow window) gcontext last pos `thenIO_`
            handleEvent pos
          _                  -> handleEvent last
  in
  handleEvent (XPoint 0 0)
```

2.2 Displays and Screens

An X session begins by making a connection with the X server. `XOpenDisplay` takes a server name and returns an X display object as the client's handle on the server.

```
xOpenDisplay :: String -> IO XDisplay
```

In X, a display can conceptually support many screens.

```
xDisplayRoots :: XDisplay -> [XScreen]
```

Functions below extract screen attributes. Every screen provides a root window (the whole screen), on which the X Window tree hierarchy is built.

```
xScreenBlackPixel :: XScreen -> XPixel
xScreenWhitePixel :: XScreen -> XPixel
xScreenRoot       :: XScreen -> XWindow
```

2.3 Windows

Besides two required arguments (parent and size), `xCreateWindow` takes a list of optional arguments of type `XWinAttribute`.

```
xCreateWindow :: XWindow -> XRect -> [XWinAttribute] -> IO XWindow

data XRect         = XRect  Int Int Int Int      -- x, y, width, height

data XPixel        = XPixel Integer              -- a 1, 2, 4, 16, or 32 bit integer

data XWinAttribute = XWinBackground XPixel
                   | XWinEventMask XEventMask
```

```
                  | ...

data XEventMask     = XEventMask [XEventMaskKey]

data XEventMaskKey = XButtonMotion   -- allow XEventMotionNotify when button is down
                   | XButtonPress    -- allow XEventButtonPress
                   | ...
```

Background color is directly represented as pixel values. **XEventMask** tells the X server what kind of events are reported to the program. The example code has **XEventMask [XButtonMotion, XButtonPress]**, which allows the server to issue **XMotionNotifyEvent** when the mouse moves with a button down, and **XButtonPressEvent** upon any mouse button press.

A window is not immediately visible once it is created. Mapping a window makes it visible.

```
xMapWindow :: XWindow -> IO ()
```

2.4 Graphics Contexts

Most graphics operations require a graphics context argument. A graphics context is a set of attributes such as color, font, and line style, etc. Like **xCreateWindow**, **xCreateGcontext** takes a list of optional attributes.

```
xCreateGcontext :: XDrawable -> [XGCAttribute] -> IO XGcontext

data XGCAttribute = XGCBackground XPixel
                  | XGCForeground XPixel
                  | ...

data XDrawable    = XDrawWindow XWindow
                  | XDrawPixmap XPixmap
```

A graphics context has to be associated with a drawable object. There are two kinds of drawable objects in X — windows and pixmaps. A pixmap can be thought of as a two-dimensional array of pixels. (Bitmap, the more familiar term, is a pixmap with single bit pixels.)

2.5 Graphic Operations

X provides basic graphics operations for drawing points, lines, rectangles, and arcs. **XDrawLine** takes a drawable object, a graphics context, and two end points. Notice that we can use **xDrawLine** to draw on both windows and pixmaps. Graphics context specifies the color, thickness, and line styles, etc.

```
xDrawLine :: XDrawable -> XGcontext -> XPoint -> XPoint -> IO ()

data XPoint = XPoint Int Int                    -- x, y
```

2.6 Events

Events are normally sent to the program by the X server. Most often, they are generated by keyboard and mouse input devices. Events occur asynchronously. Interactive X applications consist of event receiving and processing loops.

XGetEvent waits for and returns the next event.

```
xGetEvent :: XDisplay -> IO XEvent

data XEvent      = XEvent XEventType [XEventSlot]

data XEventType = XButtonPressEvent
                | XMotionNotifyEvent
                | ...

data XEventSlot = XEventPos XPoint
                | ...

xEventType :: XEvent -> XEventType
xEventPos  :: XEvent -> XPoint
```

XEvent is the type of all possible events. It has a event type and a list of slots. Selection functions extract slots of interest. For example, **xEventPos** returns the mouse pointer position. Different types of events have different slots. (The CLX manual has the details.) It is an error to extract a non-existing slot.

In the example, the **handleEvent** function is crucial and deserves a closer look.

```
  let
    handleEvent :: XPoint -> Dialogue
    handleEvent last =
      xGetEvent display `thenIO` \event ->
        let pos = xEventPos event
        in
        case (xEventType event) of
          XButtonPressEvent  -> handleEvent pos
          XMotionNotifyEvent ->
            xDrawLine (XDrawWindow window) gcontext last pos `thenIO_`
            handleEvent pos
          _                  -> handleEvent last
  in
  handleEvent (XPoint 0 0) `thenIO_`
```

Keep in mind that **XMotionNotifyEvent** only arrives when we press one of the mouse buttons (see 2.3). When the program starts and the user moves the mouse around without pressing a button, no events are generated. Once a button is pressed, **XButtonPressEvent** arrives, which tells us where the drawing should start. Drawing continues as long as **XMotionNotifyEvents** keep arriving, until the user releases the button. **HandleEvent** then waits for the user to press the button again. (This simple program does not have a way to terminate itself!)

3 The X Library

Further explanation of the X interface is not really necessary. The Haskell files defining the interface contains the data types and external function signatures needed to use the interface. The file **$HASKELL_LIBRARY/X11/xlibprims.h** contains type signatures for all X window functions. The datatypes used by the window system are in **$HASKELL_LIBRARY/X11/xlib.hs**.

4 Setup and Run

Yale Haskell is distributed in both source and binary form. The binary release is distributed either with or without the X window support preloaded. When you ftp the Haskell compiler, you must choose the binary containing X window support. The startup banner will contain **-x** when the X support is included.

Any program that uses X facilities must import **Xlib**, like in the previous example.

If **foo.hs** imports **Xlib**, **foo.hu** has to include this line:

```
$HASKELL_LIBRARY/X11/xlib.hu
```

For example, suppose **draw.hs** contains the simple example program. We set up a two-line file **draw.hu** which contains:

```
$HASKELL_LIBRARY/X11/xlib.hu
draw.hs
```

4.1 Questions and Bug Reports

Send questions and bug reports to **haskell-request@cs.yale.edu**.