The Ya[illegible]e[illegible]Lia[illegible]Re[illegible], YALEU/DCS/RR-972

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 S,

Yale Haskell builds its I/O system using a monad. The I/O monad uses a special data constructor

the result type of functions which involve the global state. Most X window functions have global eff

are only callable from the monad. The monad is described in the Haskell 1.3 I/O system proposal.

Most Haskell X functions have a CLX counterpart. We use a simple name
Haskell function **xWindowEventMask** corresponds to the CLX function **xlib**

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
is that arguments related to geometry are abstracted into the following Has

```
data XPoint = XPoint Int Int                          -- x, y
data XSize  = XSize  Int Int                          -- width, heig
data XRect  = XRect  Int Int Int Int                  -- x, y, width
data XArc   = XArc   Int Int Int Int Float Float -- x, y, width
```

Many CLX functions return either an object or "null". We introduce a Ha
purpose.

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

For example, CLX function `xlib:window-colormap` returns a window's color
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 E

The XError type and xHandleError function that were present in an earlier versio

been removed, as they have been subsumed by a more general error handling mech Haskell 1.3 I/O system.

When an X error (or other low-level Lisp error) occurs, the I/O computation fails correct way to handle these errors is to use the **`try`** function. One reason to establish a handler in this way is so that you can call **`xCloseDisplay`** and do any other cleanup associated with your application before exiting.

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`**). Typing anything on the keyboard causes the program to exit.

2.1 C

```
module Draw where
```

```
import Xlib

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

draw :: String -> IO ()
draw host =
  xOpenDisplay host >>= \ display ->
  let (screen:_) = xDisplayRoots display
      fg_color = xScreenBlackPixel screen
      bg_color = xScreenWhitePixel screen
      root = xScreenRoot screen
```

```
      xGetEvent display >>= \event ->
        let pos = xEventPos event
in
case (xEventType event) of
          XButtonPressEvent  -> handleEvent pos
          XMotionNotifyEvent ->
            xDrawLine (XDrawWindow window) gcontext last pos >>= \() ->
            handleEvent pos
          XKeyPressEvent     -> xCloseDisplay display
          _                  -> handleEvent last
  in
  putStr "Press any key to quit.\n" >>
  handleEvent (XPoint 0 0)
```

2.2 Di,asc

An X session begins by making a connection with the X server. `XOpenDisplay` ta

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 whic the X Window tree hierarchy is built.

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

2.3 Wi

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

```
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 GC

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
```

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

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 `XMotionNotifyEvent`s keep arriving, until the user releases the button. `HandleEvent` then waits for the user to press the button again, or to press a key to terminate the function call.

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 `$HASKELL_LIBRARY/X11/xlibprims.hi` contains type signatures for all X window functions. The datatypes used by the window system are in `$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 Q,e$_g$B,g Re,

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