Mousetoxin: Exceedingly verbose reimplementation of Ratpoison

Troels Henriksen (athas@sigkill.dk)

29th November 2009

Chapter 1

Introduction

This paper describes the implementation of Mousetoxin, a clone of the X11 window manager Ratpoison. Mousetoxin is implemented in Literate Haskell, and the full implementation is presented over the following pages. To quote the Ratpoison manual:

> Ratpoison is a simple Window Manager with no fat library dependencies, no fancy graphics, no window decorations, and no rodent dependence. It is largely modeled after GNU Screen which has done wonders in the virtual terminal market.
>
> All interaction with the window manager is done through keystrokes. Ratpoison has a prefix map to minimize the key clobbering that cripples EMACS and other quality pieces of software.
>
> Ratpoison was written by Shawn Betts (`sabetts@vcn.bc.ca`).

Apart from serving as a literate example, partly for me and partly for others, of how to define a practical side-effect heavy Haskell program, Mousetoxin also serves as an experiment to answer two other questions of mine:

- Can Haskell programs be written that obey common Unix principles (such as responding to `SIGHUP` by reloading their configuration file) without too much trouble, and without turning the entire program into an impure mess?
- How feasible is it to write nontrivial programs in Literate Haskell? This is partially a test of the adequacy of available Haskell tools, partially a test of Literate Programming as a concept in itself.

Chapter 2

Startup logic

The *Main* module defines the main entry point for the program when invoked, namely the function *main*. We shall do all our command-line parameter processing here, so that the rest of the program can restrict itself to the actual logic of managing windows.

```haskell
module Main (main) where

import Control.Applicative
import System.Console.GetOpt
import System.Environment
import System.Exit
import System.IO
```

We import the window management logic.

```haskell
import Mousetoxin.Config
import Mousetoxin.Core
```

The startup function extends the default (static) configuration with information gleamed from the environment, then with the values specified by the command line options (if any), and finally passes the resulting configuration to the window management logic entry point.

The `DISPLAY` environment variable is the standard method of communicating to X programs which display they should connect to. Programs such as `startx` will automatically set it appropriately before running *.xsession*/*.xinitrc* (or the program indicated on the command line), and as Mousetoxin will most likely be started as part of the general X session startup, the `DISPLAY` environment variable is where we will find out which display we should connect to. We will, however, support an explicit `--display` command-line option in case the user desires to have Mousetoxin connect to some other arbitrary display.

```haskell
main :: IO ()
main = do
  opts <- getOpt RequireOrder options <$> getArgs
  dstr <- getEnv "DISPLAY" `catch` (const $ return "")
  let cfg = defaultConfig{ displayStr = dstr }
```

```haskell
case opts of
  (opts', [], []) -> mousetoxin =<< foldl (>>=) (return cfg) opts'
  (_, nonopts, errs) -> do
    mapM_ (hPutStrLn stderr) $ map ("Junk argument: " ++) nonopts
    usage <- usageStr
    hPutStrLn stderr $ concat errs ++ usage
    exitFailure
```

Our command-line options are described as mappings from their short and long names (eg. `-h` and `--help`) to a (monadic) function that extends the Mousetoxin configuration (taking into account the option argument if there is one).

```haskell
options :: [OptDescr (WMConfig -> IO WMConfig)]
options = [optHelp, optVersion, optDisplay]
```

The `--help` option follows standard Unix convention by having the short name `-h` and immediately terminating Mousetoxin after running. The code for generating the option list is factored out into a definition by itself, because we also wish to display it if the user specifies an invalid option.

```haskell
optHelp :: OptDescr (WMConfig -> IO WMConfig)
optHelp = Option ['h'] ["help"]
  (NoArg $ \_ -> do
    hPutStrLn stderr =<< usageStr
    exitSuccess)
  "Display this help screen."

usageStr :: IO String
usageStr = do
  prog <- getProgName
  let header = "Help for " ++ prog ++ " " ++ versionString
  return $ usageInfo header options
```

The `--version` option is very similar, also terminating the program after printing the version information.

```haskell
optVersion :: OptDescr (WMConfig -> IO WMConfig)
optVersion = Option ['v'] ["version"]
  (NoArg $ \_ -> do
    hPutStrLn stderr ("Mousetoxin " ++ versionString ++ ".")
    hPutStrLn stderr "Copyright (C) 2009 Troels Henriksen."
    exitSuccess)
  "Print version number."
```

We do not care to check the format of the `--display` option parameter at this stage, as this checking will be done by much more knowledgeable code down the line, when we attempt to actually connect to the X server.

```haskell
optDisplay :: OptDescr (WMConfig -> IO WMConfig)
optDisplay = Option ['d'] ["display"]
  (ReqArg (\arg cfg -> return $ cfg{displayStr = arg}) "dpy")
  "Specify the X display to connect to."
```

Chapter 3

Mousetoxin.Core

The *Core* module concerns itself with the basics of maintaining a consistent window manager state, as well as initiating and maintaining communication with the X server.

```
module Mousetoxin.Core
  (mousetoxin
  , WM
  , WMCommand
  , cmdError
  , liftWM
  , WMState (..)
  , WMConfig (..)
  , WMSetup (..)
  , SplitType (..)
  , FrameTree (..)
  , WindowFrameTree
  , ManagedWindow (..)
  , isManaged
  , isDisplayed
  , withDisplay
  , frameByPath
  , changeAtPath
  , findFramePath
  , leafPaths
  , changeFrames
  , focusOnWindow
  , focusOnFrame
  , focusWindowPair
  , focusWindowNumber
  , focusWindow
  , otherWindow
  , grabKeys
  , message
  , createOverlay
  , installSignalHandlers
  , uninstallSignalHandlers
```

```
  , spawnChild
  , rootMask
  , clientMask
  , scanWindows
  , evalCmdString
  , CommandArg (..)
  , consumingArgs
  , accepting
  , PlainString (String)
  , PlainInteger (Integer)
  , WMWindow (Window)
  , withGrabbedKeyboard
  , readKey
  , getInput
  , doGetInput
  , EditCommand (..)
  , EditorState (..)
  , EditorContext (..)
  , CommandResult (..)
  ) where
```

We use the X11 library for communicating with the X server. X11 is an FFI-wrapper around Xlib and its interface closely mirrors the C library. While this is potentially more brittle than an X library written purely in Haskell, it permits much easier adaptation of new improvements to the Xorg X server, as focus will be on supporting new server extensions in Xlib or other libraries written in C. Besides, I am not aware of any implementation of the X11 protocol written in Haskell. Note that we hide the name *refreshKeyboardMapping* from *Graphics.X11.Xlib*; this is because it is also defined in the *Graphics.X11.Xlib.Extras* module.

```
import Graphics.X11.Xlib hiding (refreshKeyboardMapping)
import Graphics.X11.Xlib.Extras
import Graphics.X11.Xlib.Font
import Graphics.X11.Xlib.Misc
import Graphics.X11.Xlib.Cursor
```

Additionally, we will make use of a range of various utility modules.

```
import Control.Applicative
import Control.Arrow
import Control.Concurrent
import Control.Exception (finally)
import Control.Monad.CatchIO hiding (bracket)
import Control.Monad.Error
import Control.Monad.Reader
import Control.Monad.State
import Data.Bits
import qualified Data.Foldable as F
import Data.List
import qualified Data.Map as M
import Data.Maybe
```

```
import Data.Monoid
import Data.Ord
import Data.Time.Clock
import Foreign.C.String
import System.IO
import System.Posix.IO
import System.Posix.Process
import System.Posix.Signals
import System.Posix.Types
```

3.1 Frame Representation

First, we will define a representation for the division of the visible screen estate into *frames*. The default Mousetoxin setup is to show a single full-screen window, but it can at any time be split either horizontally (resulting in top-bottom stacking) or vertically (resulting in left-right), repeated indefinitely. We represent this frame configuration as a tree: a node can be either a leaf (a frame possibly containing a window), a horizontal split, or a vertical split. Splits do not always divide the screen space equally, and we must represent this as well. Additionally, it will be convenient if we only use a single data constructor for both kinds of splits, with an extra value (of type *SplitType*) indicating the direction of the split. We lose no expressive power this way, as we can easily pattern-match on both the constructor and this value.

We opt to store the two children of a split along with two *proportions* that indicate the relative space allocation. For example, a 25/75 split between the two children of a vertical split might be represented by *Split Vertical* (1, *child1*) (3, *child2*). The values convey no information apart from their relative sizes, which means we do not have to adjust the frame tree if the screen resolution changes. On the other hand, if we need to make a modification such as "increase the size of the left child of the split by ten pixels (and reduce the right by the same)," then we first need to calculate the dimensions of the current split in pixels. On the other hand, we don't need to convert back: a proportion of 600/200, representing 600 pixels to the left window and 200 to the right, is perfectly valid, and identical to 3/1.

The *FrameTree* type is parametrised on the type of its leaves, but this is only so we can easily define instances for type-classes. In practice, we will only use the *WindowFrameTree* type.

```
data SplitType = Vertical
  | Horizontal
data FrameTree w = Frame (Maybe w)
  | Split SplitType (Integer, FrameTree w) (Integer, FrameTree w)
type WindowFrameTree = FrameTree Window
```

We will often need to index, or change, the frame tree at certain locations, and we therefore need a way to address nodes within it. As each node can be a leaf or a split, we can create a simple and lightweight address type as a list of *Either* () ()-values. A *Left* value will indicate the left branch of a split, a *Right* value the right branch, and the end of the list will indicate the current node.

```
type FrameRef = [Either () ()]
```

We define a function for retrieving the window (if any) at a given position in the tree.

$$\begin{array}{l}
\mathit{frameByPath} :: \mathit{FrameTree}\ w \rightarrow \mathit{FrameRef} \rightarrow \mathit{Maybe}\ w \\
\mathit{frameByPath}\ (\mathit{Frame}\ w)\ [\,] = w \\
\mathit{frameByPath}\ (\mathit{Split}\ _\ (_, w)\ _)\ (\mathit{Left}\ _ : \mathit{rest}) = \mathit{frameByPath}\ w\ \mathit{rest} \\
\mathit{frameByPath}\ (\mathit{Split}\ _\ _\ (_, w))\ (\mathit{Right}\ _ : \mathit{rest}) = \mathit{frameByPath}\ w\ \mathit{rest} \\
\mathit{frameByPath}\ _\ _ = \mathit{Nothing}
\end{array}$$

Another function will be used for changing the tree at a given location — it can be expected that most of these changes will involve changing the window at a leaf, but we can handle changes in an arbitrary location.

$$\begin{array}{l}
\mathit{changeAtPath} :: \mathit{FrameTree}\ w \\
\quad \rightarrow \mathit{FrameRef} \\
\quad \rightarrow (\mathit{FrameTree}\ w \rightarrow \mathit{FrameTree}\ w) \\
\quad \rightarrow \mathit{FrameTree}\ w \\
\mathit{changeAtPath}\ (\mathit{Split}\ \mathit{skind}\ (x, w)\ (y, w'))\ (\mathit{Left}\ _ : \mathit{rest})\ f = \\
\quad \mathit{Split}\ \mathit{skind}\ (x, (\mathit{changeAtPath}\ w\ \mathit{rest}\ f))\ (y, w') \\
\mathit{changeAtPath}\ (\mathit{Split}\ \mathit{skind}\ (x, w)\ (y, w'))\ (\mathit{Right}\ _ : \mathit{rest})\ f = \\
\quad \mathit{Split}\ \mathit{skind}\ (x, w)\ (y, (\mathit{changeAtPath}\ w'\ \mathit{rest}\ f)) \\
\mathit{changeAtPath}\ v\ _\ f = f\ v
\end{array}$$

We shall also have need of a function for finding the path to a given window. Most notably, this will be needed when a window is destroyed, as we will have to remove it from the frame setup (and possibly replace it with another). Note that this definition makes use of the fact that *Maybe* is an applicative functor: the trick to the function is the <|> operator, which in this case will return its left-hand side if it's a *Just*-value, it's right-hand side otherwise.

$$\begin{array}{l}
\mathit{findFramePath} :: \mathit{Eq}\ w \Rightarrow \mathit{FrameTree}\ w \rightarrow w \rightarrow \mathit{Maybe}\ \mathit{FrameRef} \\
\mathit{findFramePath}\ (\mathit{Split}\ _\ (_, w)\ (_, w'))\ e = \\
\quad (\mathit{Left}\ ():) <\$> \mathit{findFramePath}\ w\ e \\
\quad <|> (\mathit{Right}\ ():) <\$> \mathit{findFramePath}\ w'\ e \\
\mathit{findFramePath}\ (\mathit{Frame}\ (\mathit{Just}\ w))\ e \mid e \equiv w = \mathit{Just}\ [\,] \\
\quad \mid \mathit{otherwise} = \mathit{Nothing} \\
\mathit{findFramePath}\ _\ _ = \mathit{Nothing}
\end{array}$$

For some user commands, we will need to enumerate (or iterate across) all possible paths to leaves in a frame tree, so we define a function to return these. Note that we also return paths leading to empty frames.

$$\begin{array}{l}
\mathit{leafPaths} :: \mathit{FrameTree}\ w \rightarrow [\mathit{FrameRef}] \\
\mathit{leafPaths}\ (\mathit{Frame}\ _) = [[\,]] \\
\mathit{leafPaths}\ (\mathit{Split}\ _\ (_, w)\ (_, w')) = \\
\quad \mathit{map}\ (\mathit{Left}\ ():)\ (\mathit{leafPaths}\ w) +\!\!+\ \mathit{map}\ (\mathit{Right}\ ():)\ (\mathit{leafPaths}\ w')
\end{array}$$

Finally, let us define the function fulfilling the very purpose of the *FrameTree*: allocating screen real estate to windows. In order to be general, we define a function that calculates space for all *nodes* of the frame tree (except for the root), even those that are splits or empty frames. For a given *FrameTree*, provided with a tuple specifying the available space, we wish to obtain a list mapping each node to its screen location (upper-left corner) and dimensions. We use floating-point math to split the available space, but are careful to ensure that we don't lose pixels to rounding errors. At worst, a frame may be given a pixel or two more than the specified proportions might suggest, but even that is unlikely that the low numbers we are working with.

```
renderFrames :: FrameTree w
  → (Dimension, Dimension)
  → [(FrameTree w, ((Position, Position), (Dimension, Dimension)))]
renderFrames f d = (f, ((0, 0), d)) : children f
  where children (Split skind (x, w) (y, w')) =
    let split = fromIntegral x / fromIntegral (x + y) :: Double
      ldim   = truncate $ split ∗ fromIntegral (acc d)
      rdim   = acc d − ldim
      lefts  = renderFrames w $ (dmut ∘ const) ldim d
      rights = renderFrames w' $ (dmut ∘ const) rdim d
    in lefts ++ map (translate $ fromIntegral ldim) rights
      where translate = second ∘ first ∘ pmut ∘ (+)
        (acc, pmut, dmut) = case skind of
          Vertical → (fst, first, first)
          Horizontal → (snd, second, second)
    children _ = [ ]
```

It is now easy to define a function that finds the position and size of all windows contained in a frame tree. We merely call *renderFrames* and extract a window from all *Frame* nodes that contain a window, discarding the rest.

```
allocateSpace :: FrameTree w
  → (Dimension, Dimension)
  → [(w, ((Position, Position), (Dimension, Dimension)))]
allocateSpace f d = concat $ map windowsOf $ renderFrames f d
  where windowsOf (Frame (Just w), ds) = [(w, ds)]
    windowsOf _ = [ ]
```

For future convenience, we also define the membership of *FrameTree* in various type-classes.

```
instance Functor FrameTree where
  fmap f (Frame (Just w)) = Frame (Just $ f w)
  fmap _ (Frame Nothing) = Frame Nothing
  fmap f (Split _ (x, w) (y, w')) = Split Vertical (x, fmap f w) (y, fmap f w')
instance F.Foldable FrameTree where
  foldMap f (Frame (Just w)) = f w
  foldMap _ (Frame Nothing) = mempty
  foldMap f (Split _ (_, w) (_, w')) = F.foldMap f w `mappend` F.foldMap f w'
```

3.2 Dynamic Window Manager State

The *Window* type by itself is a handle to a resource in the X-server, and does not contain all the information that we wish to bind to the notion of a window in the Mousetoxin-sense. Therefore we define the *ManagedWindow* data type, which will contain not just the Xlib window handle, but also miscellaneous information supporting various Mousetoxin features. The fields of *ManagedWindow* will be described in detail when they are used. We will still store plain *Window*s in the frame tree, as various operations will change the fields in *ManagedWindow*, and we do not want to update the leaves of the frame tree every time this happens. For simplicity, our aim is that non-core code should not ever have to deal with plain *Window* values, but rather only operate on *ManagedWindow* via helper functions.

```
data ManagedWindow = ManagedWindow
  { window          :: !Window
  , accessTime      :: !UTCTime
  , windowWMTitle   :: !String
  , expectedUnmaps  :: !Integer
  , pointerCoords   :: (Position, Position)
  , sizeHints       :: SizeHints
  }
```

A *ManagedWindow* is uniquely identified by the *Window* handle it contains.

```
instance Eq ManagedWindow where
  x ≡ y = window x ≡ window y
```

The *expectedUnmaps* field of *ManagedWindow* is necessary for properly handling a subtlety of window management and will be described in detail later on.

We store the mutable window manager state in the *WMState* data structure; this is the only data structure that will be changed as a result of command invocation by the user. At present, *WMState* stores a map of window handles to managed windows structures. The assignment of numeric identifiers to windows is purely a user-centric notion, but it is still of critical importance to ensure that no two windows ever share a number, which the use of a *Map* will help with. The *WMState* structure contains a number of fields that are exclusively used by specific commands and facilities; these will be described in detail at their actual point of use.

```
data WMState = WMState
  { managed     :: !(M.Map Integer ManagedWindow)
  , frames      :: !WindowFrameTree
  , focusFrame  :: !FrameRef
  , childProcs  :: !(M.Map ProcessID (ProcessStatus → WM ()))
  }
```

3.3 Static configuration

The user-provided program configuration is stored in a *WMConfig* structure; this won't normally change during program execution, but it might if the Mousetoxin configuration file is reloaded. In Mousetoxin, keys are bound to *command strings*, consisting of a command name and (optionally) arguments. The implementation of command execution is described in Section 3.8 on page 32, while specific commands are covered in Chapter 5 on page 45.

```
data WMConfig = WMConfig
  { displayStr   :: !String
  , prefixKey    :: !(KeyMask, KeySym)
  , keyBindings  :: !(M.Map (KeyMask, KeySym) String)
  , commands     :: !(M.Map String (WMCommand ()))
  , editCommands :: !(M.Map (KeyMask, KeySym) (EditCommand CommandResult))
  , overlayBorderWidth :: !Dimension
  , overlayPadding :: !(Dimension, Dimension)
  }
```

The *WMSetup* data structure primarily contains information that is immutable throughout the entire execution of Mousetoxin, namely the connection to the X11 server, and the root window of the screen (implying that we support only a single screen per server). We also store the user configuration here. Despite the fact that it is possible for the configuration to change, such changes are rare and only happen between commands.

```
data WMSetup = WMSetup
  { display  :: Display
  , rootW    :: !Window
  , overlayWindow :: !Window
  , gcontext :: !GC
  , config   :: WMConfig
  }
```

3.4 The *WM* Monad

All Mousetoxin commands execute in the *WM* monad, which incorporates error signalling, a read-only *WMSetup*, a mutable *WMState*, and finally wraps around the *IO* monad, as communication with the X-server is an inherently impure I/O operation. Note that a great number of typeclass instances are derived - while Haskell 98 supports only a small, predefined set (*Eq*, *Ord*, *Enum*, *Bounded*, *Show*, *Read*), the Glasgow Haskell extension *GeneralizedNewtypeDeriving* allows derivation of even user-defined classes. Additionally, we shall later see that it is convenient to define special-purpose monads that wrap around *WM*, yet still occasionally have to perform *WM* operations. Hence, we define the *MonadWM* typeclass and the *liftWM* method for lifting *WM* operations, exactly the same way *liftIO* does.

```
class (Monad m) => MonadWM m where
  liftWM :: WM a -> m a
newtype WM a = WM (ErrorT String (ReaderT WMSetup (StateT WMState IO)) a)
  deriving (Functor, Monad, MonadIO, MonadCatchIO, MonadState WMState,
    MonadReader WMSetup, MonadError String)
runWM :: WMSetup -> WMState -> WM a -> IO (a, WMState)
runWM c st (WM a) = do (r, s) <- runStateT (runReaderT (runErrorT a) c) st
  case r of
    Left e -> fail $ "Unhandled Mousetoxin error: " ++ e
    Right v -> return (v, s)
```

The *ErrorT* monad transformer is quite nifty, as it allows us to make use of an exception-like mechanism without using IO actions. Its error information is a simple string, and whenever an error is signalled, the end result will be that the error message is displayed to the user in the overlay window via the *message* function described in the next section.

```
wmError :: String → WM a
wmError = throwError
```

Note that we also derive *WM* as an instance of *MonadCatchIO*, yet there's no such monad in the constructor! *MonadCatchIO* is a thin wrapper that works for any *MonadIO*, and which enables the use of the "generic" I/O exception handling mechanisms defined in *Control.Monad.CatchIO*, which are thin wrappers around *Control.Exception*. We can't use the latter in Mousetoxin, as they assume you only wish to protect *IO* actions, and not any impure monad, such as *WM*.

There is a single remaining problem, however. Consider the convenient function *bracket*, which encapsulates a common idiom in programming: acquire a resource, do something with it, then release it. *bracket* ensures that the final cleanup step is done, even if the middle step causes an exception (in some object-oriented imperative languages, this might be referred to as a "finally"-block). The *bracket* in *Control.Monad.CatchIO* does not handle the fact that the *WM* monad has its own pure error mechanism (from *ErrorT*), however, and will not run the cleanup code if we trigger an error through *wmError*. Fortunately, it is easy to use the general mechanisms to define our own *bracket*.

```
bracket :: WM a → (a → WM b) → (a → WM c) → WM c
bracket before after thing = block $ do
  a ← before
  r ← unblock (thing a)
    `onException` after a
    `catchError` λe → after a ≫ throwError e
  after a
  return r
```

3.5 Manipulating the window manager state

It is of great importance that we maintain the integrity of our data structures, and to facilitate this we will ensure that all manipulation happens through a small set of invariant-preserving functions.

Whenever we are informed about a new window, we will have to find and assign it a number. As we prefer small numbers, this will be done by iterating upwards from zero until we find one not already used to identify a window. We are not likely to ever be managing a very large set of windows, so this function should remain fast.

```
freeWindowNum :: WM (Maybe Integer)
freeWindowNum = do
  m ← gets managed
  let inUse x = elem x $ M.keys m
  return $ find (¬ ∘ inUse) [0 . .]
```

The X server does not know about our *ManagedWindow* structure and will only tell us about plain *Windows*. Hence, we need a function from which we can get the *ManagedWindow* corresponding to a given *Window* (if any). We might as well return the window number too.

```
managedWindow :: Window → WM (Maybe (Integer, ManagedWindow))
managedWindow win = find ((≡) win ∘ window ∘ snd) <$> allmanaged
  where allmanaged = M.toList <$> gets managed
```

A window is managed if and only if there is a *ManagedWindow* entry for it.

```
isManaged :: Window → WM Bool
isManaged = liftM isJust ∘ managedWindow
```

A window is displayed if and only if it is present in a leaf of the frame tree.

```
isDisplayed :: ManagedWindow → WM Bool
isDisplayed w = elem (window w) <$> F.toList <$> gets frames
```

The *withDisplay* function is a small utility for making code that accesses the display value more aesthetically pleasing.

```
withDisplay :: (Display → WM a) → WM a
withDisplay f = asks display >>= f
```

For manipulating the frame tree, we provide *changeFrames*. The central idea is that we accept a pure function taking a *WindowFrameTree* and returning a new *WindowFrameTree*, after which we communicate with the X server to implement the changes in the tree (moving and resizing windows to their new positions), as well as ensuring that whichever window (if any) that now occupies the focus frame will have focus by the X server. Apart from the frame tree itself, we also pass the path to the focus frame, and expect it to be similarly updated. This maintains the invariant that the focus frame path at all times refers to a valid frame in the tree.

```
changeFrames :: ((WindowFrameTree, FrameRef)
  → (WindowFrameTree, FrameRef))
  → WM ()
changeFrames f = do
  oldtree   ← gets frames
  oldfocus ← gets focusFrame
  oldfocusw ← focusWindow
  let (newtree, newfocus) = f (oldtree, oldfocus)
  if (validPath newfocus newtree)
    then do modify (λs → s{frames = newtree
        , focusFrame = newfocus})
        layoutWindows oldtree newtree
        newfocusw ← focusWindow
        when (newfocusw ≢ oldfocusw) $ do
          maybe (return ()) lostFocus oldfocusw
          setFocusWindow newfocusw
    else wmError "Invalid frame reference."
  where validPath p t = elem p $ leafPaths t
```

When a window loses focus we store the position of the mouse cursor. When (if) it regains focus, we will restore this saved position.

```
lostFocus :: ManagedWindow -> WM ()
lostFocus mw = withDisplay $ \dpy -> do
  rootw <- asks rootW
  (_, _, _, x, y, _, _, _) <- liftIO $ queryPointer dpy rootw
  updateWindowData (window mw) $ \w ->
    w{pointerCoords = (fromIntegral x, fromIntegral y)}
```

The function *layoutWindows* is responsible for moving and resizing the X windows in response to frame tree changes. We pass both the old and the new tree, so that we only have to take actual changes into account.[1] We also hide every window that was visible in the old tree, but not in the new one, while indiscriminately mapping every window visible in the new tree. This is not a problem, as mapping is a no-op on an already mapped window.

```
layoutWindows :: WindowFrameTree -> WindowFrameTree -> WM ()
layoutWindows from to = withDisplay $ \dpy -> do
  rootw <- asks rootW
  wa <- liftIO $ getWindowAttributes dpy rootw
  let allocs = allocateSpace to
    (fromIntegral $ wa_width wa,
      fromIntegral $ wa_height wa)
  forM_ allocs $ \(win, ((x, y), (w, h))) -> do
    mwin <- managedWindow win
    maybe (return ())
      (layoutWindow (x + fromIntegral (wa_x wa),
        y + fromIntegral (wa_y wa))
        (w, h)) $ snd <$> mwin
  mapM_ hideWindow delta
  where delta = F.toList from \\ F.toList to
```

Laying out a single window involves fitting it within the frame allocated to it by Mousetoxin. While the frame itself is inflexible, we have a lot of leeway with regards to whether or not the window takes up all the space allocated to it, a decision that is based on the *sizeHints* value associated with the window.

```
layoutWindow :: (Position, Position)
  -> (Dimension, Dimension)
```

[1] You may have noticed the somewhat unappealing use of *fromIntegral* in *layoutWindows* (and other functions). This is necessary for converting 32-bit signed numbers from the window attributes (the *CInt* type) to the 32-bit unsigned numbers we use to indicate window dimensions (the *Dimension* type, which is actually a *Word32*). This pattern will be repeated relatively often, and we should consider whether any bugs could be introduced this way.

In practice, we will never use unsigned values with magnitudes that cannot be represented in a signed value of equal size, implying that unsigned to signed conversions will not result in truncation or overflow.

The safety of signed-to-unsigned conversion is not as simple, unfortunately: negative signed values do appear, notably in window positions (a window with a negative position would be offset above or to the left of the physical screen border), but it turns out that all our conversions involving negative numbers are about converting *CInts* to *Int32s*, both simple 32-bit signed integers. Whenever we convert a signed integer to an unsigned integer, such as the expression *fromIntegral $ wa_width wa* in *changeFrames*, we are guaranteed that the signed integer is positive; in this case because a window dimension is always > 0.

```
    → ManagedWindow
    → WM ()
layoutWindow (x, y) (w, h) mwin = withDisplay $ λdpy → do
  liftIO $ do moveResizeWindow dpy win x' y' w' h'
    mapWindow dpy win
    raiseWindow dpy win
    where win = window mwin
      ((w', h'), (x', y')) = constrainSize (sizeHints mwin) ((w, h), (x, y))
```

We perform a little sanity-checking on the size hints so that a bad-behaved client will not cause us to crash. When adjusting the size of a window such that it is smaller than its enclosing frame, we have to decide how to align it relative to the empty space. When the adjustment is caused by a maximum size or aspect ratio limitation, it makes sense to centre the window in the frame, but when we adjust due to size increment considerations, keeping an upper-left alignment will result in the most aesthetic layout.

```
constrainSize :: SizeHints
        → ((Dimension, Dimension), (Position, Position))
        → ((Dimension, Dimension), (Position, Position))
constrainSize sh =
    maybe id constrainToMax (sh_max_size sh)
  ∘ maybe id constrainToMin (sh_min_size sh <|> sh_base_size sh)
  ∘ maybe id constrainToAspect (sh_aspect sh)
  ∘ maybe id constrainToInc  (liftM2 (,)
                                (sh_resize_inc sh)
                                (sh_base_size sh <|> sh_min_size sh <|> Just (0,0)))
    where constrainToMax (mw, mh)
        | min mw mh > 0 = centre (min mw, min mh)
        | otherwise = id
      constrainToMin (mw, mh) = centre (max mw, max mh)
      constrainToInc ((iw, ih), (bw, bh))
        | min iw ih > 0 = topleft (max bw ∘ λw → w − w `mod` iw,
          max bh ∘ λh → h − h `mod` ih)
        | otherwise = id
      constrainToAspect ((minx, miny), (maxx, maxy)) x@((w, h), _)
        | or [minx < 1, miny < 1, maxx < 1, maxy < 1] = x
        | w ∗ maxy > h ∗ maxx = centre (const $ h ∗ maxx `div` maxy, id) x
        | w ∗ miny < h ∗ minx = centre (id, const $ w ∗ miny `div` minx) x
        | otherwise = x
      centre (fw, fh) ((w, h), (x, y)) =
        ((fw w, fh h),
          (x + (fromIntegral $ (w − fw w) `div` 2),
            y + (fromIntegral $ (h − fh h) `div` 2)))
      topleft (fw, fh) ((w, h), p) = ((fw w, fh h), p)
```

As a great deal of our interaction will be about changing (or removing) the window displayed in the focus frame, we define a convenience function. We also take care to ensure that the same window does not appear twice in the frame tree, this is accomplished by changing the focus frame path, rather than the frame tree itself, if we are asked to change focus to an already visible window.

$$
\begin{aligned}
&\mathit{focusOnWindow} :: \mathit{Maybe}\ \mathit{ManagedWindow} \rightarrow \mathit{WM}\ ()\\
&\mathit{focusOnWindow}\ \mathit{mwin} = \mathbf{do}\\
&\quad \mathit{ff} \leftarrow \mathit{gets}\ \mathit{focusFrame}\\
&\quad \mathit{fs} \leftarrow \mathit{gets}\ \mathit{frames}\\
&\quad \mathbf{case}\ \mathit{mwin} \gg\!= \mathit{findFramePath}\ \mathit{fs} \circ \mathit{window}\ \mathbf{of}\\
&\qquad \mathit{Just}\ \mathit{path} \rightarrow \mathit{changeFrames}\ \$\ \lambda(t, \_) \rightarrow (t, \mathit{path})\\
&\qquad \mathit{Nothing} \rightarrow \mathit{changeFrames}\ \$\ (\lambda(t, f) \rightarrow\\
&\qquad\quad (\mathit{changeAtPath}\ t\ \mathit{ff}\ \$\ \mathit{const}\ (\mathit{Frame}\ \$\ \mathit{mwin} \gg\!= \mathit{return} \circ \mathit{window}), f))
\end{aligned}
$$

Also for convenience, we define functions to return the window, number, or both, in the frame that has focus, if any. Note that the inner **do**-block in *focusWindowPair* is actually in the *Maybe* monad.

$$
\begin{aligned}
&\mathit{focusWindowPair} :: \mathit{WM}\ (\mathit{Maybe}\ (\mathit{Integer}, \mathit{ManagedWindow}))\\
&\mathit{focusWindowPair} = \mathbf{do}\ \mathit{WMState}\{\mathit{frames} = \mathit{fs}\\
&\quad , \mathit{focusFrame} = \mathit{ff}\} \leftarrow \mathit{get}\\
&\quad \mathbf{case}\ \mathit{frameByPath}\ \mathit{fs}\ \mathit{ff}\ \mathbf{of}\\
&\qquad \mathit{Just}\ \mathit{win} \rightarrow \mathit{managedWindow}\ \mathit{win}\\
&\qquad \mathit{Nothing} \rightarrow \mathit{return}\ \mathit{Nothing}\\
&\mathit{focusWindowNumber} :: \mathit{WM}\ (\mathit{Maybe}\ \mathit{Integer})\\
&\mathit{focusWindowNumber} = \mathit{liftM}\ (\mathit{liftM}\ \mathit{fst})\ \mathit{focusWindowPair}\\
&\mathit{focusWindow} :: \mathit{WM}\ (\mathit{Maybe}\ \mathit{ManagedWindow})\\
&\mathit{focusWindow} = \mathit{liftM}\ (\mathit{liftM}\ \mathit{snd})\ \mathit{focusWindowPair}
\end{aligned}
$$

Changing the focused frame is not as simple as merely changing the *focusFrame* field in the *WMState*-structure: we also have to make sure that the input focus is properly set.

$$
\begin{aligned}
&\mathit{focusOnFrame} :: \mathit{FrameRef} \rightarrow \mathit{WM}\ ()\\
&\mathit{focusOnFrame}\ \mathit{newfocus} = \mathbf{do}\\
&\quad \mathit{modify}\ (\lambda s \rightarrow s\{\mathit{focusFrame} = \mathit{newfocus}\})\\
&\quad \mathit{setFocusWindow} =\!\ll \mathit{focusWindow}
\end{aligned}
$$

Many user-commands will want to interact with the most recently accessed window not currently being displayed — the so-called *other window*.

$$
\begin{aligned}
&\mathit{otherWindow} :: \mathit{WM}\ (\mathit{Maybe}\ \mathit{ManagedWindow})\\
&\mathit{otherWindow} = \mathbf{do}\\
&\quad \mathit{others} \leftarrow (\mathit{filterM}\ \mathit{undisplayed} \circ M.\mathit{elems}) =\!\ll \mathit{gets}\ \mathit{managed}\\
&\quad \mathit{return}\ \$\ \mathit{listToMaybe}\ \$\ \mathit{reverse}\ \$\ \mathit{sortBy}\ (\mathit{comparing}\ \mathit{accessTime})\ \mathit{others}\\
&\quad \mathbf{where}\ \mathit{undisplayed} :: \mathit{ManagedWindow} \rightarrow \mathit{WM}\ \mathit{Bool}\\
&\qquad \mathit{undisplayed} = (\mathit{liftM}\ \neg \circ \mathit{isDisplayed})
\end{aligned}
$$

One of the most important occurrences during the execution of Mousetoxin will be the creation of new windows. The *manageWindow* function will be called whenever a window is created. When we take over management of a new window, we display it in the focus frame, replacing whatever was already there. Additionally, we express to the server that we should be notified of *PropertyNotify*-events in the new window - this allows us to be notified whenever window properties, such as the window title, changes (see Section 3.7 on page 24). Mousetoxin relies on a globally available set of keyboard commands, so we also have to *grab* the prefix key on the window, such that we, and not the window itself, will be informed whenever it is pressed (see Section 3.7 on page 30).

```
manageWindow :: Window → WM ()
manageWindow w = withDisplay $ λdpy → do
  num ← freeWindowNum
  case num of
    Just k → do
      mw ← makeManagedWindow w
      modify (λs → s{managed = M.insert k mw (managed s)})
      liftIO $ do mapWindow dpy w
        selectInput dpy w clientMask
      grabKeys w
      focusOnWindow $ Just mw
    Nothing → wmError "could not allocate window number"
clientMask :: EventMask
clientMask = propertyChangeMask
  .|. focusChangeMask
```

Constructing a managed window structure is a fairly mechanical process wherein we request a few bits of information about the window. Of course, there is no guarantee that this data will be constant throughout the lifetime of the window, which is why we'll have to detect when it changes and update. This is described later on, in Section 3.7 on page 24.

The concept of window size hinting, which we interact with through *getWMNormalHints*, is part of the ICCCM, and many windows may not implement it. Fortunately, the X11 library interfaces seems to provide us with a set of harmless default values in such cases.

```
makeManagedWindow :: Window → WM ManagedWindow
makeManagedWindow w = withDisplay $ λdpy → do
  s ← getWindowName w
  t ← liftIO $ getCurrentTime
  h ← liftIO $ getWMNormalHints dpy w
  rootw ← asks rootW
  (_, _, _, x, y, _, _, _) ← liftIO $ queryPointer dpy rootw
  return ManagedWindow{ window = w
    , accessTime           = t
    , windowWMTitle        = s
    , expectedUnmaps = 0
    , pointerCoords        = (fromIntegral x, fromIntegral y)
    , sizeHints            = h}
```

The ICCCM defines the `WM_NAME` property for specifying the user-visible name of a window. We cannot expect that all windows have this property, however, in which case we return a placeholder string. Note that we are forced to do this by way of handling an IO exception, as even if we check whether the property exists before calling *getTextProperty*, the asynchronous nature of X makes it possible that the property has been removed by the time we actually make the call.

```
getWindowName :: Window → WM String
getWindowName w = withDisplay $ λdpy → liftIO $
```

```
    (peekCString =<< (liftM tp_value $ getTextProperty dpy w wM_NAME))
    `Prelude.catch` (const $ return "Unnamed Window")
```

We may sometimes need to update the information stored in a managed window; and in these cases we will usually only have a plain *Window* value to go by.

```
updateWindowData :: Window → (ManagedWindow → ManagedWindow) → WM ()
updateWindowData w f = do
  ws ← gets managed
  case find ((≡) w ∘ window ∘ snd) $ M.toList ws of
    Just (num, mw) → do
      modify (λs → s{ managed = M.insert num (f mw) ws})
    Nothing → return ()
```

As far as the window manager state is concerned, unmanaging a window consists of removing it from the list of managed windows, and possibly removing it from the frame tree. A window can become unmanaged in many ways (the owner unmapping it, the user asking Mousetoxin to close it), but the details of these cases are handled in their individual implementations. As empty screen space is wasted space, when a visible window is unmanaged, we would like another window to take its place. For this, we select the *other window*, the last accessed undisplayed window.

```
unmanageWindow :: ManagedWindow → WM ()
unmanageWindow mwin = do
  modify (λs → s{ managed = M.fromList
    [(k, mwin') | (k, mwin') ← M.toList $ managed s
      , mwin' ≢ mwin]})
  path ← liftM2 findFramePath (gets frames) $ return (window mwin)
  ow ← otherWindow
  when (isJust path) $
    changeFrames (λ(t, f) →
      (changeAtPath t (fromJust path) $ const (Frame $ window < $ > ow), f))
```

When Mousetoxin's window layout changes such that some window is no longer visible, we have to hide it. We could also leave it in place, under the assumption that it has been hidden because some other window now obscures its place on the screen, but it might still be visible if the new window is not perfectly rectangular, or has transparent parts. Also, if the frame tree is changed later on, the new layout of windows may not fully obscure a "hidden" window that takes up the entire root window. This could be alleviated by never permitting empty frames in the frame tree if there is an available window, but adding such an esoteric restriction due to a technical issue would be poor design. Instead, we unmap windows that we do not intend to be visible at the moment.

There is a subtlety with unmapping windows, however. Normally, when we are informed that a window has been unmapped, we unmanage it, as we assume the owner no longer intends that the user be able to interact with it. The problem is that when we unmap a window ourselves, we will receive the same notification, with no apparent way to distinguish between an application-initiated unmapping and a Mousetoxin-initiated unmapping. The common hack to solve this problem is to keep a counter of *expected unmaps* – we increment the counter when hiding a window, and check it when we receive an unmap notification event. If the counter is zero, we unmanage the window, otherwise we decrement the counter and do nothing else.

```
hideWindow :: Window → WM ()
hideWindow win = withDisplay $ λdpy → do
  t ← liftIO $ getCurrentTime
  updateWindowData win $
    λmwin → mwin{ expectedUnmaps = expectedUnmaps mwin + 1
      , accessTime = t}
  liftIO $ unmapWindow dpy win
```

Mousetoxin responds to user requests through an *overlay window*, a simple, unmanaged window that typically appears in a corner, and is no larger than necessary for the information within it. For convenience and uniformity, we define a function for mapping, resizing, and raising the overlay. X does not permit zero-dimension windows, so we ensure that the overlay is at least a single pixel in width and height (not counting any border). We also ensure that the overlay never starts to the left of the physical screen: we cannot prevent an information loss if we are asked to display a wider overlay than the screen has room for, but we can at least ensure that the starting (leftmost) information is visible, as it is usually the most important. The *withOverlay* function clears any already existing graphics in the overlay window, and it is thus not possible to use successive calls to build up an overlay window in parts; all drawing has to be within the scope of a single call to *withOverlay*.

```
withOverlay :: (Dimension, Dimension) → (Window → WM a) → WM a
withOverlay (width, height) f = withDisplay $ λdpy → do
  border ← overlayBorderWidth < $ > asks config
  let screen = defaultScreenOfDisplay dpy
    swidth = widthOfScreen screen
    w      = max 1 $ fromIntegral width
    h      = max 1 $ fromIntegral height
    xpos = max 0 $ fromIntegral $ swidth − width − border ∗ 2
  overlay ← asks overlayWindow
  liftIO $ do
    setWindowBorderWidth dpy overlay border
    moveResizeWindow dpy overlay xpos 0 w h
    mapRaised dpy overlay
    clearWindow dpy overlay
    sync dpy False
  f overlay
```

We also supply a simple function for hiding the overlay window if we desire to get rid of it for some reason.

```
clearOverlay :: WM ()
clearOverlay = withDisplay $ λdpy → do
  overlay ← asks overlayWindow
  liftIO $ do
    lowerWindow dpy overlay
    moveResizeWindow dpy overlay 0 0 1 1
    setWindowBorderWidth dpy overlay 0
    sync dpy False
```

Input focus can be directed to a managed window, or to no window (*no focus* to the user, for example when we give focus to a frame that does not contain a window). A naive implementation would be to grant focus to the root window, but this turns out to be a bad idea. Consider how keypress events work: when they fire, the X server will create a list of all mapped windows containing the mouse pointer position, and deliver the event to *the topmost window that is a child of the focus window* that a program is listening for keypress events on. As almost all programs listen for keypresses on their own main window, which is a child of the root window, the effective result of granting input focus to the root window is a "focus follows mouse"-policy.

Fortunately, Mousetoxin has its own window - the *overlay window*, which we normally make use of for user interaction, though it also serves as a convenient key sink. As we control the overlay window, we can ensure that it will never react badly to keypress events – in fact, we will never use the *selectInput* function to express interest in keypress events on it. See Section 3.7 on page 24 for more details on events and event listening.

As a basic UI principle, we also warp the pointer to the pointer coordinates of the window at the time it was last visible.

```haskell
setFocusWindow :: Maybe ManagedWindow → WM ()
setFocusWindow w = withDisplay $ λdpy →
  case w of
    Just (ManagedWindow{ window = w', pointerCoords = (x, y)}) → do
      rootw ← asks rootW
      liftIO $ do setInputFocus dpy w' revertToPointerRoot 0
        warpPointer dpy 0 rootw 0 0 0 0 x y
    Nothing → do overlay ← asks overlayWindow
        liftIO $ setInputFocus dpy overlay revertToPointerRoot 0
```

For easily conveying textual information to the user, we provide an Emacs-style *message* function. An overlay window will pop up with the passed string, and it will stay until dismissed. The *message* function should support displaying multiple lines, and two obvious strategies suggest themselves: accept a list of strings as the message (where each string would be interpreted as a line by itself), or accept a string containing newline characters (`'\n'` in Haskell).

We opt for the latter strategy, as the former has the problem that the passed strings might themselves contain newline characters, and it is not obvious what to do with them (we might use them for breaking the message into even more lines, or just ignore them altogether). By the principle of least surprise, we choose the most intuitive interface, even if it means the ability to handle multiple lines is no longer directly expressed in the type signature of the function. Note that the Xlib function *drawString* cannot handle newline characters on its own, and we have to break the string into lines anyway.

drawString draws the string with the baseline at the given coordinates, so while we advance line-by-line down the overlay, we have to subtract the descent of the font to find the position we should actually draw at. See Figure 3.1 on the following page for additional information about text positioning.

```haskell
message :: String → WM ()
message msg = withDisplay $ λdpy → do
  gc ← asks gcontext
```

When computing the vertical space necessary for displaying a line of characters in a given font, we normally do not pay attention to the contents of the actual string. Doing so might make the line "jump" if the content changes from having no tall characters, such as an *h*. Instead, we use the sum of the maximum ascent and descent from the baseline any string can have in the font.

```
fontHeight :: FontStruct → Dimension
fontHeight f = fromIntegral $
  ascentFromFontStruct f
   + descentFromFontStruct f
```

Figure 3.1: Text height

```
font ← liftIO $ fontFromGC dpy gc ≫= queryFont dpy
(xpad, ypad) ← overlayPadding < $ > asks config
let ss      = splitLines msg
    width   = xpad ∗ 2 + fromIntegral (foldl max 0 $ map (textWidth font) ss)
      -- fromIntegral is safe because length is always >=0.
    height  = fontHeight font ∗ (fromIntegral ◦ length) ss + ypad ∗ 2
    descent = descentFromFontStruct font
    leftpad = fromIntegral xpad
    toppad = fromIntegral ypad
withOverlay (width, height) $ λw → liftIO $ do
  forM_ (zip ss $ map (∗fontHeight font) [1 ..]) $ λ(s, y) → do
    drawString dpy w gc leftpad (fromIntegral y − descent + toppad) s
where splitLines :: String → [String]
  splitLines s = case break (≡ '\n') s of
    (s′, []) → [s′]
    (s′, _ : ss) → s′ : splitLines ss
```

Overlay windows are important to Mousetoxin — apart from the important main overlay used for window listing and command input, we use temporary overlays for many small tasks. For example, when a frame receives focus, we show a small overlay window within it to make it clear where the focus has gone. We desire somewhat uniform appearance for these overlay windows, so we define the function *createOverlay*. Here, we create an unmapped 1x1 window with zero border width located in the upper left corner of the screen.

```
createOverlay :: Display → Screen → Window → IO Window
createOverlay dpy screen root = do
  let visual = defaultVisualOfScreen screen
    attrMask = cWOverrideRedirect
        .|. cWBackPixel
        .|. cWBorderPixel
    black    = blackPixelOfScreen screen
    white    = whitePixelOfScreen screen
  allocaSetWindowAttributes $ λattrs → do
    set_override_redirect attrs True
```

$set\_background\_pixel\ attrs\ white$
$set\_border\_pixel\ attrs\ black$
$createWindow\ dpy\ root$
$\quad 0\ 0\ 1\ 1\ 0$ -- x, y, width, height, border
$\quad copyFromParent$
$\quad inputOutput$ -- class
$\quad visual\ attrMask\ attrs$

$createMainOverlay :: Display \rightarrow Screen \rightarrow Window \rightarrow IO\ Window$
$createMainOverlay = createOverlay$

3.6 Signals and events

Mousetoxin receives input from two asynchronous sources:

- Unix signals (see Section 3.6 on the next page).
- Events from the X-server.

There are two ways of waiting for both of these occurrences at the same time: polling and using two threads, both of which block for input. We opt for the latter option, using a thread that does nothing but handle signals (the Haskell system actually does this for us) and one that reads events from the connection to the X server.

As we still wish to keep our actual logic single-threaded, we will have to make use of an *MVar* to communicate with the main thread. Note that the actual *MVar* will have to be stored in the *WM* monad, as have to create it as an actual IO operation. Short of *unsafePerformIO*, we cannot have global names bound to *MVar*s as global names are not evaluated in the *IO* monad.

Communication consists of exchanging actions to be run in the WM monad.

$mkSyncVar :: IO\ (MVar\ (WM\ ()))$
$mkSyncVar = newEmptyMVar$

We are only interested in `SIGPIPE` and `SIGCHLD` (see), so the signal handler is installed as follows. We ignore `SIGPIPE`, the signal sent when trying to write to a dead program[2], as the default response to this signal is to terminate the program, and we don't want to die if one of our children stop unexpectedly.

$installSignalHandlers :: MonadIO\ m \Rightarrow MVar\ (WM\ ()) \rightarrow m\ ()$
$installSignalHandlers\ var = liftIO\ \$\ \mathbf{do}$
$\quad installHandler\ openEndedPipe\ Ignore\ Nothing$
$\quad installHandler\ sigCHLD\ (Catch\ \$\ handleSIGCHLD\ var)\ Nothing$
$\quad return\ ()$

We do not want our child processes to inherit our signal handlers (especially not the one for `SIGPIPE`), so we define a function for turning them off.

$uninstallSignalHandlers :: MonadIO\ m \Rightarrow m\ ()$
$uninstallSignalHandlers = liftIO\ \$\ \mathbf{do}$

[2]This is a simplification.

```
    installHandler openEndedPipe Default Nothing
    installHandler sigCHLD Default Nothing
    return ()
```

Subprocess management

Apart from being a window manager, Mousetoxin also functions as a launcher for various programs (see the *exec* command in Chapter 5 on page 45). In Unix, a child process that terminates will still be retained in the system as a *zombie* unless the parent process invokes the (blocking) `waitpid` system call, but we really don't want to spawn a new thread for every child process just to call `waitpid`. The solution is to establish a signal handler for `SIGCHLD`, which is the signal sent by the operating system when a child process terminates. We could get rid of the zombie processes by indicating that we wish to ignore `SIGCHLD`, but we can do better than that: when the user starts a program, he might be interested in its return value (in particular if it errors out), or perhaps just be notified that it has stopped. Hence, we define a facility for associating a (child) process ID with an action that will be performed when the process terminates.

We store mappings from *ProcessID*s to *WM* actions in the *childProcs* field of the window manager state. As a `SIGCHLD` does not tell us *which* process terminate, we have to iterate through the entire set and check whether each is still alive, performing the associated action if not.

```
reapChildren :: WM ()
reapChildren = do
  children ← filter′ check =≪ gets childProcs
  modify $ λs → s{ childProcs = children }
    where filter′ f = liftM M.fromList ∘ filterM f ∘ M.toList
      check (pid, a) = do
        s ← liftIO $ getProcessStatus False False pid
        maybe (return True) (λs′ → a s′ ≫ return False) s
```

When a SIGCHLD is received we ask the main thread to call *reapChildren*.

```
handleSIGCHLD :: MVar (WM ()) → IO ()
handleSIGCHLD var = putMVar var reapChildren
```

As a final convenience, we define a simple interface for starting subprocesses.

```
spawnPID :: String → WM ProcessID
spawnPID x = liftIO ∘ forkProcess ∘ finally nullStdin $ do
  createSession
  uninstallSignalHandlers
  executeFile "/bin/sh" False ["-c", x] Nothing
where
  nullStdin = do
    fd ← openFd "/dev/null" ReadOnly Nothing defaultFileFlags
    dupTo fd stdInput
    closeFd fd
spawnChild :: String → (ProcessStatus → WM ()) → WM ()
```

$$
\begin{array}{l}
\mathit{spawnChild}\ x\ a = \mathbf{do} \\
\quad \mathit{pid} \leftarrow \mathit{spawnPID}\ x \\
\quad \mathit{modify}\ \$\ \lambda s \rightarrow s\{\mathit{childProcs} = M.\mathit{insert}\ \mathit{pid}\ a\ \$\ \mathit{childProcs}\ s\}
\end{array}
$$

3.7 Talking to X

The *mousetoxin* function serves as the starting point of the module. Connecting to the X server is trivial (though we must be aware that the connection, as with all network communication, can fail), and does not merit further discussion. But let us consider our next step: in order to manage windows, Mousetoxin will need to be informed whenever certain events happen on the X display.

Window creation and destruction: Whenever a new window is created, we must be informed so that we can handle it appropriately (at least by adding it to our own set of managed windows). Likewise, we must also be informed whenever a window is destroyed. Information about window destruction is conveyed through *DestroyNotify*, but we will not use the parallel *CreateNotify* event, as many applications may create windows that are not going to be shown on the screen. Instead, we shall make use of X11's notion of *mapping*: a window starts out unmapped (invisible), and the process of making a window visible on the screen is referred to as mapping it. We will listen to the *MapRequest* event, so we have a chance of refusing attempts to take focus. The *UnmapNotify* event is sent whenever a window is unmapped.

Change in window configuration: Many programs will attempt to change the size or position of their root window, something that conflicts with the basic intent of Mousetoxin. We shall intercept the *ConfigureRequest* window event to prevent such attempted operations. The related *ResizeRequest* event seems to cover a subset of the functionality provided by *ConfigureRequest*, and we therefore do not need to capture it.

Mouse button presses: Whenever a mouse button is pressed, we wish to shift focus to the layout frame containing the window receiving the mouse press event, before passing on the event. We thus have to capture *ButtonPress* events. We also opt to capture *ButtonRelease* events for the same reason.

Keyboard presses: As Mousetoxin is solely controlled via the keyboard, we conceptually have to inspect all key input to see whether it's part of a Mousetoxin command invocation, before passing it on to the window that has focus. We opt to capture only *KeyPress*, not *KeyRelease*, as we have no need of the latter. Also, in practice, we don't inspect all input ourselves, but rather use a so-called *passive grab* in the X server, described in more detail in Section 3.7 on page 30.

Changes in window properties: A number of important values attached to windows, such as their title, is defined by a *window property*. We must be informed whenever some of these properties (again, such as the window title) changes. The X server will convey this information to us through the *PropertyNotify* event (called *PropertyEvent* in the Haskell Xlib wrapper).

We can be informed of some of these events by setting an event mask for the root window of the screen, while for others we have to set an event mask for each individual managed window. In the former category, *DestroyNotify* and *UnmapNotify* are associated with *StructureNotifyMask*; and *MapRequest* and *ConfigureRequest* with *SubstructureRedirectMask*. In the latter category, we have *ButtonPress* and *ButtonRelease* with *ButtonPressMask* and *ButtonReleaseMask* respectively; and finally *PropertyNotify* with *PropertyChangeMask*. These masks (as the name implies) are merely bit-patterns, and we can use a standard binary or-operation to combine them. Only a single client can select for *SubstructureRedirectMask*, so if an existing window manager is already running, we will receive an error.

```
rootMask :: EventMask
rootMask = substructureNotifyMask
  .|. substructureRedirectMask
  .|. propertyChangeMask
  .|. buttonPressMask
```

The code for opening and initialising the display is straightforward, though obscured by the fact that *selectInput* makes use of the Xlib error handling mechanism, while *openDisplay* throws normal Haskell IO exceptions. At this point, any error causes a catastrophic exit.

```
setupDisplay :: String → IO Display
setupDisplay dstr = do
  dpy ← openDisplay dstr `Prelude.catch` λ_ → do
    error $ "Cannot open display '" ++ dstr ++ "'."
  let dflt = defaultScreen dpy
  rootw ← rootWindow dpy dflt
  setErrorHandler $ error "Another window manager is already running."
  selectInput dpy rootw rootMask
  sync dpy False
  return dpy
```

The function *mousetoxin* uses *setupDisplay* to connect to the X server, after which it sets up an appropriate environment for executing the *WM* monad, and starts the main event-handling loop. Almost every function run by Mousetoxin will be in response to an event handled by this loop, with the sole exception of asynchronous Unix signals. We have to select *KeyPress* events on the overlay window (via *KeyPressMask*) as much of our eventual input handling will be done by giving the overlay window focus.

We disable the Xlib error handling mechanism, as we can ensure there are no real errors on our own, and we would otherwise crash hard with a *BadWindow* error when unmapping a destroyed window.

```
mousetoxin :: WMConfig → IO ()
mousetoxin conf = do
  dpy ← setupDisplay $ displayStr conf
  let screen = defaultScreenOfDisplay dpy
    rootw = rootWindowOfScreen screen
  overlay ← createMainOverlay dpy screen rootw
  selectInput dpy overlay keyPressMask
```

```
xSetErrorHandler
mapWindow dpy overlay
gc ← createGC dpy rootw
setFont dpy gc =<< fontFromFontStruct <$> loadQueryFont dpy "9x15bold"
let cf = WMSetup
      {display       = dpy
      , rootW         = rootw
      , overlayWindow = overlay
      , gcontext      = gc
      , config        = conf
      }
    st = WMState
      {managed    = M.empty
      , frames     = Frame Nothing
      , focusFrame = []
      , childProcs = M.empty
      }
runWM cf st $ do grabKeys rootw
  setFocusWindow Nothing
  scanWindows
  wmMainLoop
return ()
```

The main loop of Mousetoxin consists of continuously taking and executing *WM* actions from an *MVar* that is in return filled by signal handlers and a thread that receives events from the X-server.

The only slightly tricky thing is this latter thread, as we cannot naively call *nextEvent* to block while waiting for input from the server. It appears that *nextEvent* applies a signal mask, meaning that our signal handlers will be delayed until after the next event has been received from the server. Hence, we only call *nextEvent* if we have made sure that there are events already in the queue (checked with *pending*). If there are no such events, we do our own blocking on the file descriptor representing our connection to the server, as that will not cause trouble with respect to signals.

```
wmMainLoop :: WM ()
wmMainLoop = withDisplay $ λdpy → do
  var ← liftIO $ mkSyncVar
  let getAndHandle = do   -- Loop body.
        (join $ liftIO (takeMVar var))
          `catchError` λerr →
            when (err ≢ "") $ message err
        liftIO $ sync dpy False
  liftIO $ forkIO $ allocaXEvent $ λev → forever $ do
    liftIO $ sync dpy False
    cnt ← pending dpy
    when (cnt ≡ 0) $
      threadWaitRead $ Fd $ connectionNumber dpy
    e ← nextEvent dpy ev ≫ getEvent ev
    putMVar var $ handleEvent e
  installSignalHandlers var
```

```
  message $ "Welcome to Mousetoxin!"
  forever getAndHandle
```

When we start running, any number of windows may already be children of the root window. We should manage all these as if they had been newly created.

```
scanWindows :: WM ()
scanWindows = withDisplay $ λdpy → do
  rootw ← asks rootW
  (_, _, wins) ← liftIO $ queryTree dpy rootw
  mapM_ manageWindow =≪ filterM ok wins
  where ok win = withDisplay $ λdpy → do
    wa ← liftIO $ getWindowAttributes dpy win
    a ← liftIO $ internAtom dpy "WM_STATE" False
    p ← liftIO $ getWindowProperty32 dpy a win
    let ic = case p of
      Just (3 : _) → True
      _ → False
    return $ ¬ (wa_override_redirect wa)
      ∧ (wa_map_state wa ≡ waIsViewable ∨ ic)
```

We come now to the *handleEvent* function, which takes an *Event* and responds appropriately.

MapRequestEvent is sent for newly created children of the root window ("newly started programs"), or any other unmapped child of the root, when it is mapped. Some windows, those that have their *override_redirect* bit set, do not send requests when they are mapped, but just map immediately. Incidentally, windows that are *override_redirect* are also the only windows that we should never manage. Hence, if we receive a *MapRequestEvent* for a window that we do not currently manage, we should start managing it. We may occasionally receive mapping requests for windows that we do manage, but that we have unmapped to hide them. These requests are ignored, effectively meaning that it is impossible for an existing window to steal focus. Only the creation of new windows, or user action, can change which windows appear on the screen.

Note that *MapRequestEvent*s are not generated when we ourselves use the *mapWindow*-function, only when other programs try to map a window.

```
handleEvent :: Event → WM ()
handleEvent (MapRequestEvent{ ev_window = w }) = do
  client ← isManaged w
  when (¬ client) $
    manageWindow w
```

An application may unmap its windows at any time for any reason, but Mousetoxin will always respond by unmanaging the window, removing all stored information, and freeing up its window number. If the application decides to remap the window later on, it will be assigned a new window number. This policy means we do not have to care about window creation: as a window manager, we are concerned with windows that applications intend to be visible (mapped), not any child window of the root that may be created for utility reasons.

Not all *UnmapEvent*s are caused by the application; Mousetoxin unmaps windows that are not visible in the current frame tree. We keep track of this in the *expectedUnmaps* field of the *ManagedWindow* structure, and do not unmanage the window if the unmapping was requested by Mousetoxin.

```
handleEvent (UnmapEvent{ev_window = win}) = do
  mwin ← managedWindow win
  case mwin of
    Just (_, mwin'@(ManagedWindow{expectedUnmaps = 0})) →
        -- Unexpected unmap, so unmanage.
      unmanageWindow mwin'
    Just (_, _) → updateWindowData win $ λmwin' →
        -- This unmap was Mousetoxin-initiated, so just reduce counter.
      mwin'{expectedUnmaps = expectedUnmaps mwin' − 1}
    Nothing → return ()
```

While window creation is not our concern, we still have to react to window destruction, as they do not cause an unmap event to be sent. A window being destroyed means we have to unmanage it immediately.

```
handleEvent (DestroyWindowEvent{ev_window = win}) = do
  mwin ← managedWindow win
  fromMaybe (return ()) (unmanageWindow ∘ snd < $ > mwin)
```

*ConfigureRequestEvent*s are sent in a similar fashion to *MapRequestEvent*s, when programs that are not us try to change the configuration (size, position, border, etc) of a window. Any window managed by us will already have its proper position (or get it soon) as determined by *layoutWindows*, but some programs may assume that their configure requests are granted, which may result in visual artifacts. To ensure that windows know their proper size, we make sure that they receive a reconfigure event containing their actual dimensions. We assume that non-managed windows have a good reason for being so, and grant their configuration request.

```
handleEvent e@(ConfigureRequestEvent{ev_window = w}) = withDisplay $ λdpy → do
  client ← managedWindow w
  case client of
    Nothing → liftIO $
      configureWindow dpy w (ev_value_mask e) $ WindowChanges
        {wc_x            = ev_x e
        , wc_y           = ev_y e
        , wc_width       = ev_width e
        , wc_height      = ev_height e
        , wc_border_width = 0
        , wc_sibling     = ev_above e
        , wc_stack_mode  = ev_detail e}
    Just _ → do
      wa ← liftIO $ getWindowAttributes dpy w
      liftIO $ configureWindow dpy w mask $ WindowChanges
        {wc_x            = ev_x e
        , wc_y           = ev_y e
        , wc_width       = ev_width e
        , wc_height      = ev_height e
        , wc_border_width = 0
```

```
          , wc_sibling      = ev_above e
          , wc_stack_mode = ev_detail e}
        liftIO $ sync dpy False
        changeFrames id
    liftIO $ sync dpy False
        where mask = ev_value_mask e .&. (1 .|. 2 .|. 4 .|. 8 .|. 16)

handleEvent e@(MappingNotifyEvent{ev_window = win})
    | ev_request e ≡ mappingKeyboard = do
    rootw ← asks rootW
    client ← isManaged win
    liftIO $ refreshKeyboardMapping e
    when (client ∨ win ≡ rootw) $ grabKeys win
```

For some reason the *FocusIn* and *FocusOut* events lack their own value constructors in the Xlib binding, and are instead represented as *AnyEvent*s with specific event types set. They have no data payload, so it does not matter. A focus change may mean that we have lost our passive keyboard grab, which we must then reestablish to prevent the user from losing the ability to communicate with Mousetoxin.

```
handleEvent (AnyEvent{ev_window = win, ev_event_type = etype})
    | etype ≡ focusIn ∧ etype ≡ focusOut = do
    client ← isManaged win
    when (client) $ grabKeys win
```

When we receive a keypress, we check whether it is the prefix key, and if so, ask for another key for the actual command. We will only receive *KeyEvent*s for the keys that we have explicitly grabbed, and we don't technically have to check whether the key is the prefix key if that's all we grab. Yet, there is no harm in being cautious.

```
handleEvent (KeyEvent{ev_event_type = t, ev_state = m, ev_keycode = code})
    | t ≡ keyPress = withDisplay $ λdpy → do
      s ← liftIO $ keycodeToKeysym dpy code 0
      prefix ← liftM prefixKey $ asks config
      when ((cleanMask m, s) ≡ prefix) dispatchCommand

handleEvent (PropertyEvent{ev_window = w, ev_atom = atom})
    | atom ≡ wM_NAME = do
      s ← getWindowName w
      updateWindowData w (λmw → mw{windowWMTitle = s})
    | atom ≡ wM_HINTS = withDisplay $ λdpy → do
      h ← liftIO $ getWMNormalHints dpy w
      updateWindowData w (λmw → mw{sizeHints = h})
```

With the masks we have chosen, we will also receive some extra events that we do not care about, such as *ConfigureEvent*s for our managed windows and *KeyEvent*s for key releases. We will silently ignore these.

```
handleEvent _ = return ()
```

Keymasks (information about the modifiers active when a key is pressed) can be quite complicated due to the device-independence of the X11 protocol. We define a function to filter away modifiers that we are not interested in.

```
cleanMask :: KeyMask → KeyMask
cleanMask km = complement (numLockMask
  .|. lockMask) .&. km
  where numLockMask :: KeyMask
    numLockMask = mod2Mask
```

Figure 3.2: Keymask manipulation

Input handling

Keyboard input is delivered to Mousetoxin in the form of *KeyEvent* values. The two most important values contained in the event is the *keycode*, a device-specific value identifying the pressed key, and the *keymask*, a device-independent value indicating which modifiers (such as Shift, CTRL, or Meta/Alt) were active when the key was pressed. We can convert keycodes to *keysyms*, device-independent key identifiers.

The foundation on which the entire Mousetoxin input mechanism is built on is the *passive grab* we establish on all managed windows. All keypresses will be delivered normally to the window that has focus, except for our prefix key. Thanks to the passive grab, it will be delivered to Mousetoxin as a *KeyEvent*.

```
grabKeys :: Window → WM ()
grabKeys win = do
  WMSetup{display = dpy, config = cfg} ← ask
  let (mask, sym) = prefixKey cfg
  liftIO $ ungrabKey dpy anyKey anyModifier win
  kc ← liftIO $ keysymToKeycode dpy sym
  when (kc ≢ '\0') $ liftIO $ grabKey dpy kc mask win True grabModeAsync grabModeAsync
```

Whenever we ask the user for keyboard input, we have to grab global keyboard focus in the X server. As it turns out, what we really want in those cases is to do a general "input grab" (though less severe than a real server grab, which stops *all* event processing) where the user cannot interact with any program but Mousetoxin. It is important that we make to make sure that no error or IO exception can cause us to skip releasing the focus, as the X server could otherwise be left in a confusing state. We pass the overlay window to *grabKeyboard*, which has the effect of causing all keyboard events to be reported with respect to the overlay. This is the reason we had to select *KeyPress* events on the overlay window during initialisation.

```
withGrabbedKeyboard :: WM a → WM a
withGrabbedKeyboard body = withDisplay $ λdpy → do
  WMSetup{overlayWindow = overlay} ← ask
  cursor ← liftIO $ createFontCursor dpy xC_icon
  let grabkey = do r ← liftIO $ grabKeyboard dpy overlay False
```

```
                 grabModeSync grabModeAsync
                 currentTime
                 when (r ≢ grabSuccess) $
                   wmError $ "Could not obtain keyboard grab." ++ show r
    ungrabkey = liftIO $ ungrabKeyboard dpy currentTime
    grabptr   = do r ← liftIO $ grabPointer dpy overlay True 0
                 grabModeAsync grabModeAsync
                 none cursor currentTime
                 when (r ≢ grabSuccess) $
                   wmError "Could not obtain mouse grab."
    ungrabptr = liftIO $ ungrabPointer dpy currentTime
  bracket grabkey (const ungrabkey) $ λ_ →
    bracket grabptr (const ungrabptr) (const body)
```

As Mousetoxin's user interface is wholly keyboard-based, the fundamental interaction with the user is through reading keyboard input. The function *readKey* encapsulates the logic of requesting a single keystroke from the user. We are only interested in *real* keys: we are not interested in whether the user presses a modifier key; indeed, it would be very poor UI if the user could not even press the Shift or Control key without it being treated as a command invocation by Mousetoxin. Therefore, we loop until we receive a key that is not a modifier (as determined by the Xlib function *isModifierKey*).

```
readKey :: WM ((KeySym, KeyMask), String)
readKey = withDisplay $ λdpy → do
  (s, m) ← liftIO $ allocaXEvent $ λev → do
    let readKey′ = do
      maskEvent dpy keyPressMask ev
      res ← keysymFromEvent ev
      case res of
        Just ((s, m), str) → if (isModifierKey s)
          then readKey′
          else return ((s, m), str)
        Nothing → readKey′
    readKey′
  return (s, m)
```

The function for extracting keysyms from keypress events is somewhat ugly, though unfortunately more due to design flaws in the Xlib library, than due to any inherent complexity in the process. We extract an *Event* value from the *XEventPtr* to get at the keymask and keycode of the event, yet we have to cast the *XEventPtr* to an *XKeyEventPtr* in order to use *lookupString* to extract a keysym from the event. Alternatively, we could use *keycodeToKeysym* directly, but we would have to implement the index logic (which is used to switch between the keysym maps based on whether modifier keys like Shift are pressed) ourselves. We sacrifice the aesthetics of our code in order to make use of Xlib's hopefully correct implementation of modifier logic. The returned string is the character sequence (if any) corresponding to the key, for example "a" for the a key.

```
keysymFromEvent :: XEventPtr → IO (Maybe ((KeySym, KeyMask), String))
keysymFromEvent ev = do
```

```
    et ← get_EventType ev
    if (et ≡ keyPress) then do
      e ← getEvent ev
      case e of
        (KeyEvent{ ev_state = m })
            → do (ks, str) ← lookupString $ asKeyEvent ev
              case ks of
                Just s → do
                  return $ Just
                    (if (m .&. shiftMask ≢ 0)
                      then (s, cleanMask m `xor` shiftMask)
                      else (s, cleanMask m),
                      str)
                Nothing → return Nothing
        _ → return Nothing   -- Should never happen.
      else return Nothing
```

Let us now define a function that does the following.

1. Reading a keypress from the keyboard.
2. Checking whether said key is bound to a command string.
3. If so, evaluate that command string.

```
dispatchCommand :: WM ()
dispatchCommand = do
  WMConfig{ keyBindings = bindings } ← asks config
  clearOverlay
  ((s, m), _) ← withGrabbedKeyboard readKey
  case M.lookup (m, s) bindings of
    Just cmd → evalCmdString cmd
    Nothing → message $ "Key '" ++ keysymToString s ++ "' is not bound to a command."
```

3.8 Command execution

Commands are the central user-oriented mechanism for interacting with Mousetoxin and are invoked through *command strings*. Commands can take any number of parameters, and all (or some) may be provided in the command string that invoked the command. Any unsupplied parameters must be obtained through interactive querying of the user. We would like to hide this complexity from command definitions, making it completely transparent whether a parameter was provided in the command string or through an interactive query.

To accomplish this, commands are represented via the *WMCommand* monad, which wraps a *String* state around an inner *WM* monad. This string state contains the parts of the command string following the actual command; completely unprocessed, such that the command can implement whatever syntax is appropriate. For the vast majority of commands, this will be a sequence of whitespace-separated tokens. Indeed, the reason for this design is that we can write monadic actions for retrieving argument values that inspect (and consume) the string state, or interactively query the user if the string is empty. A lifting function, *liftWM*, is provided for running code the wrapped *WM* monad. This could also have been implemented by making the functions in Section 3.5 on page 12 operate on a monad transformer class, and making *WMCommand* an instance of that class, but I judge that *liftWM* is a more lightweight solution for our purposes. Indeed, it is likely that most commands will merely use *WMCommand* for the previously mentioned argument processing facilities, then perform most of their logic in a lifted monadic *WM* action.

```
newtype WMCommand a = WMCommand (StateT String WM a)
  deriving (Functor, Monad, MonadIO, MonadState String,
    MonadError String)
instance MonadWM WMCommand where
  liftWM a = WMCommand (lift a)
runWMCommand :: String → WMCommand a → WM a
runWMCommand s (WMCommand a) = runStateT a s ≫= return ∘ fst
cmdError :: MonadWM m ⇒ String → m a
cmdError = liftWM ∘ wmError
```

The format of a command string is exceedingly simple: the command consists of all characters up to the first space character, the arguments consist of all remaining characters (including this first space). A command must contain at least a single character; therefore the empty string is not a valid command string, nor is any string starting with a space character.

```
evalCmdString :: String → WM ()
evalCmdString s = do
  WMConfig{commands = cmds} ← asks config
  let (cmd, args) = break (≡ ' ') s
  when (cmd ≡ "") $ wmError "No command provided."
  case M.lookup cmd cmds of
    Just cmd′ → do
      liftIO $ putStrLn $ "!!!!!!!!!!!running command " ++ cmd
      runWMCommand args cmd′
    Nothing → wmError $ "Unknown command '" ++ cmd ++ "'."
```

Interactive Editing

We will shortly begin to discuss code that performs an interactive dialogue with the user by prompting for input. Before that is possible, however, we will have to define at least the programming interface with which the input editor will expose itself to the world. We will not descripe the actual implementation, instead delegating this to Section **??** on page ??.

The Mousetoxin input editor is the program responsible for providing a user with a way to enter and edit single-line text. It provides a simple Emacs-like user interface to the user and supports programmatic input completion. In the following, we will use *input buffer* (or just *buffer*) to refer to the actual text being edited, and *editing point* to the position in the input buffer at which further character insertions will take place. The term editing point (or just *point*) comes from Emacs, and is perhaps more commonly known as the *caret* or *editing cursor*.

The main function has the following type.

$$\begin{aligned}&getInput :: Window\\&\quad \to (CompleteRequest \to WM\ CompleteResponse)\\&\quad \to (FinishRequest \to WM\ (FinishResponse\ a))\\&\quad \to WM\ (EditResult\ a)\end{aligned}$$

The specified window is used for displaying the state of the editing process (that is, drawing the input buffer and cursor position). Two functions are provided to support tab-completion and final transformation of the input into a real value.

A completion request simply contains the string of text to be completed.

$$\textbf{type}\ CompleteRequest = String$$

The completion function should return a list of possible completions based on the input string. The completions should be *complete*, that is, start with the text passed in via the *CompleteRequest*. This means we don't have to spend time cutting off prefixes, but can just replace whatever part of the input buffer we asked to be completed.

$$\textbf{type}\ CompleteResponse = [String]$$

A finish request happens when the user presses the Enter key (or equivalent), signifying that the input is complete. We pass the entire contents of the input buffer as a string.

$$\textbf{type}\ FinishRequest = String$$

We must try to transform the text input (for example `"42"`) to a value of the desired type (for example an *Int*). If this is not possible, such as if the user has entered `"foo"` when we asked for an integer, we will return a *Left* value containing an explanation of what is wrong. Otherwise, we return the value wrapped in *Right*.

$$\textbf{type}\ FinishResponse\ a = Either\ String\ a$$

An edit result is either a value (as given by the *FinishResponse*), or *Nothing*. We do not distinguish between aborted editing sessions and input that merely could not be transformed into a value of the desired type, as the latter case is signalled through errors.

$$\textbf{type}\ EditResult\ a = Maybe\ a$$

The *CommandArg* typeclass

As previously mentioned, we would like command parameter acquisition to be transparent, whether it involves reading from the command string or performing an interactive query. The linchpin in this scheme is the *CommandArg* typeclass, which defines a method *accept* that returns a value of the desired type wrapped in *WMCommand*. The full story is a little more complicated, however. Consider a command that requires as input the number of a managed window: the value has the Haskell type *Integer*, but this type does not fully encompass the constraints that must be put on the input. Specifically, we must have that there is a managed window with a window number that corresponds to the returned integer. For this reason, *accept* takes as argument a *presentation type*, a Haskell value enforcing additional constraints on the value. The presentation type can also include other meta-information about how to retrieve the value, such as the prompt to be used in an interactive call. This class definition makes use of multi-parameter type-classes, an extension that is not part of Haskell 98.

```
class CommandArg pt a where
  accept :: pt → WMCommand a
```

Most, if not all, *CommandArg* instances will work by trying to consume some part of the argument string, and putting the remainder back when done. Subtle and dangerous bugs can appear if the *accept* method does not remove its consumed characters from the argument string, so we provide a function that uses type-checking to force us to supply a new remainder.

```
consumingArgs :: (String → WMCommand (a, String)) → WMCommand a
consumingArgs f = do
  (r, s) ← f =<< get
  put s
  return r
```

But we can do even better. If we take a high-level view of what we need to accomplish, we find that we need to obtain a string, check that it satisfies some property (like forming a numeral), and map the string to a value. We should abstract away the difference between extracting a string from the input string and asking the user for a string in an interactive query. The only complication is that we may wish to provide a list of possible valid values for interactive editing (to accommodate tab-completion), something that does not make sense when the input comes from the command string. On the other hand, supplying this list does no harm either. As we are already doing magic for the benefit of the common case, we also opt to strip leading leading whitespace if the argument comes from the command string.

```
accepting :: String
          → (CompleteRequest → WM CompleteResponse)
          → (FinishRequest → WM (FinishResponse a))
          → WMCommand a
accepting pstr compl finish = do
  consumingArgs $ λs →
    case break (≡ '\n') s of
      ("", _) → liftWM $ do    -- Command string is empty, do interaction.
        result ← stdGetInput pstr compl finish
        case result of
          Nothing → wmError ""
          Just v → return (v, "")
      (word, rest) → liftWM $ do    -- Get line from command string.
        fin ← finish $ dropWhile (≡ ' ') word
        val ← either wmError return fin
        return (val, drop 1 rest)
```

We make the assumption that newlines in a command string separate command arguments. It is possible to define *CommandArg* instances that do not follow this rule, but they will not be able to use *accepting*. Also note that interactive editing is inherently single-line, so we are guaranteed that we'll never be working with strings corresponding to argument commands containing newline characters, at least not if we use *accepting*.

As a concrete example of the *CommandArg* typeclass, let us define an instance of *CommandArg* for reading plain strings, with no constraints on their content. We will read up to the first newline character or the end of the string, whichever comes first. Our presentation type is a value of type *PlainString*, which contains no other information than the prompt to be used for interaction with the user.

```
data PlainString = String String
instance CommandArg PlainString String where
  accept (String pstr) = do
    consumingArgs $ λs →
      case break (≡ '\n') s of
        ("", _) → liftWM $ do
          result ← stdGetInput pstr (const $ return [ ]) (return ∘ Right)
          case result of
            Nothing → wmError ""
            Just v → return (v, "")
        (word, rest) → return (word, (drop 1 rest))
```

Or alternatively, through the use of *accepting*:

```
data PlainString = String String
instance CommandArg PlainString String where
  accept (String pstr) =
    accepting pstr (const $ return [ ]) (return ∘ Right)
```

We can now request string input by an expression such as *accept* (*String* "Enter string: "), which has the type *WMCommand String*.

We define an instance for reading plain integers that makes use of the *CommandArg PlainString String* instance above.

```
data PlainInteger = Integer String
instance CommandArg PlainInteger Integer where
  accept (Integer pstr) = cmdRead =≪ accept (String pstr)
```

We do not wish to use the default *read* function from the Haskell Prelude, as it throws IO errors if the input is malformed. As the input is from the user, syntax errors should be expected and handled gracefully. We use a wrapper around the *reads* function to accomplish this, but we have to filter away partial results where less than the entire string is consumed: in order to make sure that Mousetoxin will never silently assign an interpretation to malformed input, we will not partial input.

```
safeRead :: Read a ⇒ String → Either String a
safeRead s = case r s of
  Nothing → Left $ "Invalid input '" ++ s ++ "'."
```

```
    Just v → Right v
    where r = fmap fst ∘ listToMaybe ∘ filter (null ∘ snd) ∘ reads
  cmdRead :: Read a ⇒ String → WMCommand a
  cmdRead s = either cmdError return $ safeRead s
```

Our separation between presentation types and returned Haskell values has a subtle and intriguing benefit, namely that we can define multiple *CommandArg* instances for the same presentation type that have different return values for their *accept* method. Consider the task of asking the user for a window: sometimes we are interested in a window number, sometimes a *ManagedWindow* value. Logically, however, the sets of valid values are isomorphic, so we need only a single presentation type.

Our presentation type is simple, being just a data constructor with a *String* for the interactive prompt.

```
  data WMWindow = Window String
```

An instance that yields (*Integer*, *ManagedWindow*) values (window numbers paired with their window structures) will be shown next. It works by creating a map from strings of window numbers and window titles (preferring numbers in case of collisions) to pairs of window numbers and *ManagedWindow*s. The valid completions are the window numbers of all active windows, and finishing is a matter of using the input string to perform a lookup on the map.

```
  instance CommandArg WMWindow (Integer, ManagedWindow) where
    accept (Window pstr) = do
      ws ← liftWM $ gets managed
      let check str  = maybe (err str) Right $ M.lookup str allcompls
          err str    = Left $ "unknown window '" ++ str ++ "'"
          namecompls = M.foldWithKey
                         (λnum mwin → M.insert (windowWMTitle mwin) (num, mwin))
                         M.empty ws
          numcompls  = M.foldWithKey
                         (λnum mwin → M.insert (show num) (num, mwin))
                         M.empty ws
          allcompls  = M.union numcompls namecompls
          compl str  = filter (isPrefixOf str) $ M.keys numcompls
      accepting pstr (return ∘ compl) (return ∘ check)
```

We can easily use the *CommandArg WMWindow* instance for (*Integer*, *ManagedWindow*)s to define instances for *Integers* and *ManagedWindow*s. Unfortunately, we have to use explicit type annotations, as there is no way Haskell can otherwise choose a *CommandArg* instance, as one type variable in the returned tuple will always be free.

```
  instance CommandArg WMWindow Integer where
    accept w = fst <$> (accept w :: WMCommand (Integer, ManagedWindow))
  instance CommandArg WMWindow ManagedWindow where
    accept w = snd <$> (accept w :: WMCommand (Integer, ManagedWindow))
```

The appropriate instance for a given *use* of these instances can be automatically selected by the Haskell type inference algorithm, however.

3.9 The Input Editor

This section will discuss the implementation details of the input editor whose interface was described in Section 3.8 on page 33.

The mutable input editor state is given by a data structure in which we store the input buffer, the clipboard, and the editing point. With the following representation, the editing point may end up negative or larger than the size of the input buffer, in which case it will be constrained to the size of the input buffer. For supporting our completion UI, we (may) keep a list of the possible completions that are being iterated through. The idea is that we when first embarking upon a completion process, we complete to the first possible completion. On every subsequent (immediate) press of the completion key, we iterate through the list.

```
data EditorState = EditorState
  { inputBuffer :: String
  , clipboard    :: String
  , editingPoint :: Int
  , completing   :: Maybe [String]
  }
blankEditorState :: EditorState
blankEditorState = EditorState
  { inputBuffer = ""
  , clipboard    = ""
  , editingPoint = 0
  , completing   = Nothing
  }
```

We also maintain a read-only *editor context* containing the prompt and the function for finding completions.

```
data EditorContext = EditorContext
  { prompt    :: String
  , completer :: CompleteRequest → WM CompleteResponse
  }
blankEditorContext :: EditorContext
blankEditorContext = EditorContext
  { prompt = ""
  , completer = (const $ return [])
  }
```

The same way we execute window manager commands in the *WMCommand* monad, we wish to provide a monad for input editor commands. In this case, the cause is not the need to support different methods of input acquisition, but rather the desire to avoid passing the editor state and the function for completion around. And more importantly, we do not wish to change every single input editor command if we decide to add a new piece of read-only data to the input editing context.

```
newtype EditCommand a = EditCommand
  (ReaderT EditorContext (StateT EditorState WM) a)
  deriving (Functor, Monad, MonadIO, MonadState EditorState,
```

```
    MonadReader EditorContext)
instance MonadWM EditCommand where
  liftWM a = EditCommand (lift (lift a))
runEditCommand :: EditorState
  → EditorContext
  → EditCommand a
  → WM (a, EditorState)
runEditCommand s c (EditCommand a) = runStateT (runReaderT a c) s
```

Any command that executes must yield a value describing its overall change to the editing process, that is, changes that cannot be encapsulated merely by changing values in the *EditorState* structure). Five outcomes are possible:

- A command may *fail* because it is unable to execute. For example, a tab-completion command may find no valid completions, or a command to move the editing point forward by a character may find itself already at the end of the input buffer.
- A command may finish the editing session, starting the procedure of turning the textual input into a value of the desired type.
- The editing session can at any time be prematurely aborted, resulting in no value.
- A command may engage in a *completion session*. Whenever a command that does *not* engage in a completion session finishes, the *completing* field of the *EditorState* will be cleared.
- The majority of commands, such as moving the cursor or inserting a character, will have no particular effect on the overall editing session.

```
data CommandResult = Fail String
  | Done
  | Abort
  | No_Op
  | Completion
```

We can now define the *getInput* function. For convenience, it just wraps around a function that starts an editing session based on an already existing editor state.

```
getInput :: Window
  → (CompleteRequest → WM CompleteResponse)
  → (FinishRequest → WM (FinishResponse a))
  → WM (EditResult a)
getInput win comp = doGetInput blankEditorState context win
  where context = blankEditorContext{ completer = comp }
```

An input editing sessions starts by setting the position and dimensions of the indicated window to fit the prompt and starting input buffer contents. After this, we give the window a border, make it visible, draw its contents, and then start our main command reading loop, wherein we maintain an active keyboard grab.

This also means that the state of the X server will not change while we are doing interactive editing: no new windows will pop up, for example.

```
doGetInput :: EditorState
  → EditorContext
  → Window
  → (FinishRequest → WM (FinishResponse a))
  → WM (EditResult a)
doGetInput state cont win fin = withDisplay $ λdpy → do
  border ← overlayBorderWidth <$> asks config
  fixEditorPosition state cont win
  liftIO $ do
    setWindowBorderWidth dpy win border
    mapRaised dpy win
  redrawEditor state cont win
  withGrabbedKeyboard $
    doEditorCommands state cont win fin
```

We can also define a simple utility function for the common case where we use the overlay window, a prompt, and clear the overlay window after use.

```
stdGetInput :: String
  → (CompleteRequest → WM CompleteResponse)
  → (FinishRequest → WM (FinishResponse a))
  → WM (EditResult a)
stdGetInput pstr comp fin = do
  ow ← asks overlayWindow
  r ← doGetInput blankEditorState context ow fin
  clearOverlay
  return r
  where context = blankEditorContext{prompt = pstr
    , completer = comp}
```

Our command execution loop is mostly straightforward, consisting of the following sequence of steps:

1. Fetch keyboard invocation from user.
2. Look up command associated with invocation. If no such binding exists, go to 1.
3. Execute associated command.
4. Take a decision based on the command result:

   Done: Terminate editing yielding the result of passing the final input buffer to the finisher.

   Abort: Terminate editing, Returning *Nothing*.

   Completion: Go to 1.

   Otherwise, go to 1 with the *completion* field of the editor state cleared.

Additionally, we make sure that the position and visual representation of the editor state is up-to-date after every command invocation that does not terminate the editing session.

A final complication lies in our handling of unbound keys where no modifier keys are set – these are considered to be literal input, plain characters entered on the keyboard that should be inserted into the input buffer as a string.

```
doEditorCommands :: EditorState
  → EditorContext
  → Window
  → (FinishRequest → WM (FinishResponse a))
  → WM (EditResult a)
doEditorCommands state cont win finisher = do
  k@((s, m), _) ← readKey
  cmd ← fromMaybe (noBinding k)
    <$> M.lookup (m, s) <$> editCommands <$> asks config
  (result, state′) ← runEditCommand state cont cmd
  case result of
    Done  → do v ← finisher $ inputBuffer state′
               either wmError (return ∘ Just) v
    Abort → return Nothing
    Completion → continue state′
    Fail _ → continue state′{completing = Nothing}
    _      → continue state′{completing = Nothing}
  where noBinding ((_, m), str) = if (m ≡ 0)
    then do modify $ strInsert str
      return No_Op
    else return $ Fail "Unbound invocation"
    strInsert str s = s{inputBuffer = take p b ⧺ str ⧺ drop p b
      , editingPoint = p + length str}
      where p = editingPoint s
        b = inputBuffer s
    continue state′ = do
      fixEditorPosition state′ cont win
      redrawEditor state′ cont win
      doEditorCommands state′ cont win finisher
```

Our editor drawing algorithm is extremely simple, as we do not expect to ever have large amounts of text (indeed, our editor is single-line). We merely clear the entire window and draw the prompt and input buffer. The prompt is drawn by creating a graphics context that does an exclusive-or operation on the colour of the pixels it touches.

```
redrawEditor :: EditorState → EditorContext → Window → WM ()
redrawEditor state cont win = withDisplay $ λdpy → do
  gc ← asks gcontext
  font ← liftIO $ fontFromGC dpy gc ≫= queryFont dpy
  let ascent = ascentFromFontStruct font
    promptw = textWidth font (prompt cont)
    screen  = defaultScreenOfDisplay dpy
    cursorx = textWidth font precursor
```

```
      cursorwidth = fromIntegral $
        textWidth font (if atcursor ≡ ""
          then " "
          else atcursor)
  lgc ← liftIO $ createGC dpy win
  liftIO $ do
    setFunction dpy lgc gXxor
    setForeground dpy lgc $ whitePixelOfScreen screen
    clearWindow dpy win
    drawString dpy win gc 0 ascent (prompt cont)
    drawString dpy win gc promptw ascent (inputBuffer state)
    fillRectangle dpy win lgc (promptw + cursorx) 0 cursorwidth $ fontHeight font
    freeGC dpy lgc
    sync dpy False
    where precursor = take (editingPoint state) (inputBuffer state)
      atcursor = take 1 $ drop (editingPoint state) (inputBuffer state)
```

The *extent* of the editor is the position, width, and height of the window such that as much as possible of the input buffer is visible. After every command that may have modified the state, we should adjust the position and size of the window. There is little cleverness involved in calculating extent, consisting of merely the sum of the widths of the prompt and input buffer text, adding a space to the latter to make room for the cursor.

```
fixEditorPosition :: EditorState → EditorContext → Window → WM ()
fixEditorPosition state cont win = withDisplay $ λdpy → do
  ((x, y), (width, height)) ← editorExtent state cont
  liftIO $ moveResizeWindow dpy win x y width height

editorExtent :: EditorState
  → EditorContext
  → WM ((Position, Position), (Dimension, Dimension))
editorExtent state cont = withDisplay $ λdpy → do
  gc ← asks gcontext
  font ← liftIO $ fontFromGC dpy gc ≫= queryFont dpy
  let twidth s = fromIntegral $ textWidth font $ s
    width  = max minWidth (twidth (inputBuffer state ++ " ") + twidth (prompt cont))
    height = max 1 $ fontHeight font
    screen = defaultScreenOfDisplay dpy
    swidth = widthOfScreen screen
  return ((fromIntegral (swidth − width), 0), (width, height))
  where minWidth = 200
```

Chapter 4

Mousetoxin.Operations

As has been mentioned earlier, the fundamental design principle of Mousetoxin is robustness. As a consequence, user commands will mostly not interact directly with the window manager state as defined in Chapter 3 on page 5, but will instead make use of the facilities described in this chapter; facilities that not only take care to ensure the internal consistency of the program state, but also attempt to secure a greater degree of static safety. This is primarily achieved through wrapping the dynamically safe operations from Chapter 3 on page 5 in less powerful, but static, interfaces.

The functions defined here also take a more abstract view of the data, making it easier to work with the concepts (such as window numbers) that users and user commands think in.

```
module Mousetoxin.Operations
  (splitFocus
  , WMOperation
  , runWMOperation
  , withWindows
  , setFocusByRef
  ) where
import Mousetoxin.Core
import Control.Applicative
import Control.Monad.State
import qualified Data.Map as M
import System.IO
```

As an example of wrapping a dynamically safe function in a statically safe shell, let us consider frame splitting (Section 3.1 on page 7), in particular splitting the frame that currently has focus. While the function *changeFrames* does check that the provided new focus frame reference is correct in the new frame tree, we have to perform the check at runtime, as we cannot, in *changeFrames*, know how the frame tree will be modified. Of course, we rarely make large modifications to the frame tree. In fact, the most common operation will involve splitting a leaf of the tree either horizontally or vertically. When that happens to the focus frame, the focus frame path must be extended by a single element in order to be valid, something we can certainly enforce statically.

```
splitFocus :: SplitType → (() → Either () ()) → WM ()
splitFocus kind branch = do
  ow ← otherWindow
  changeFrames $ λ(t, ff) →
    (changeAtPath t ff $ λfr →
      Split kind (1, fr) (1, Frame $ liftM window ow)
    , ff ++ [branch ()])
```

```
newtype WindowRef = WindowRef Integer
newtype WMOperation a = WMOperation{ runWMOperation :: WM a }
withWindows :: ([WindowRef] → WMOperation ()) → WMOperation ()
withWindows f = WMOperation $
  do refs ← map WindowRef <$> M.keys <$> gets managed
    runWMOperation (f refs)
setFocusByRef :: WindowRef → WMOperation ()
setFocusByRef (WindowRef wr) = WMOperation $
  do focusOnWindow =<< M.lookup wr <$> gets managed
```

Chapter 5

Mousetoxin.Commands

```
module Mousetoxin.Commands
  (cmdNext
  , cmdPrev
  , cmdSelect
  , cmdOther
  , cmdWindows
  , cmdClose
  , cmdKill
  , cmdExec
  , cmdColon
  , cmdHoriSplit
  , cmdVertSplit
  , cmdUnsplitAll
  , cmdNextFrame
  , cmdResize
  , cmdBanish
  , cmdMeta
  , cmdTest
  , cmdTime
  , cmdQuitMousetoxin
  , cmdAbort
  , cmdNewWM
  , cmdTmpWM
  ) where

import Mousetoxin.Core
import Mousetoxin.Operations

import Control.Applicative
import Control.Concurrent
import Control.Monad.Error
import Control.Monad.State
import Control.Monad.Reader
import Data.List
import qualified Data.Map as M
import Data.Maybe
```

```
import Foreign.Storable
import Graphics.X11.Xlib
import Graphics.X11.Xlib.Extras
import System.Posix.Process
import System.Exit
import Data.Time.Clock
import Data.Time.Format
import Data.Time.LocalTime
import System.Locale

cmdNext :: WMCommand ()
cmdNext = liftWM $ do
  ws ← gets managed
  when (¬ $ M.null ws) $ do
    curr ← focusWindowNumber
    let new = fromMaybe (fst $ M.findMin ws) $ do
      curr′ ← curr
      find (>curr′) (M.keys ws)
    focusOnWindow $ M.lookup new ws

cmdPrev :: WMCommand ()
cmdPrev = liftWM $ do
  ws ← gets managed
  when (¬ $ M.null ws) $ do
    curr ← focusWindowNumber
    let new = fromMaybe (fst $ M.findMax ws) $ do
      curr′ ← curr
      find (<curr′) (reverse $ M.keys ws)
    focusOnWindow $ M.lookup new ws

cmdPrev′ :: WMCommand ()
cmdPrev′ = liftWM $ runWMOperation $ withWindows $ λwrs →
  setFocusByRef ⊥

cmdSelect :: WMCommand ()
cmdSelect = switch ‘catchError‘ (const cmdWindows)
  where switch = do mwin ← accept (Window "Switch to window: ")
    liftWM $ focusOnWindow (Just mwin)
```

The "other window" is the non-displayed window that has most recently been accessed.

```
cmdOther :: WMCommand ()
cmdOther = liftWM $ do
  other ← otherWindow
  case other of
    Just w → focusOnWindow $ Just w
    Nothing → return ()
```

```
cmdWindows :: WMCommand ()
cmdWindows = do
  liftWM $ do
    ws ← gets managed
    fw ← focusWindow
    ow ← otherWindow
    if (ws ≡ M.empty)
      then message "No managed windows "
      else let seperator w | fromMaybe False ((≡ w) < $ > fw) = "*"
                 | fromMaybe False ((≡ w) < $ > ow) = "+"
                 | otherwise = "-"
               lineFor (num, w) = show num ++ seperator w ++ windowWMTitle w
           in message $ intercalate "\n" $ map lineFor $ M.toList ws
```

The standard way to dismiss a window (possibly causing the owning program to finish) is to send it a courteous message asking it to clean up and shut down. The program may not close immediately, but rather ask the user whether to save any eventual unsaved changes, or the like. It may even refuse to close at all, in case it is in the middle of an uninterruptible operation. In any case, the following command is a highly cooperative affair, as it should be. There is a theoretical possibility that a given window does not support the window closing protocol, in which case we inform the user that the more radical *kill* command (see below) must be employed.

```
cmdClose :: WMCommand ()
cmdClose = liftWM $ withDisplay $ λdpy → do
  mw ← focusWindow
  case mw of
    Just (ManagedWindow{ window = w}) → do
      protocols ← liftIO $ getWMProtocols dpy w
      wm_delete ← liftIO $ internAtom dpy "WM_DELETE_WINDOW" False
      if wm_delete ∈ protocols then liftIO $
        allocaXEvent $ λev → do
          setEventType ev clientMessage
          wm_protocols ← internAtom dpy "WM_PROTOCOLS" False
          setClientMessageEvent ev w wm_protocols 32 wm_delete currentTime
          sendEvent dpy w False 0 ev
        else message "This window does not support the delete protocol, you have to use
    Nothing → return ()
```

An application sometimes hangs or misbehaves, in which case it may not respect the *cmdClose* function. Fortunately, Xlib provides the function *killClient*, with which we can force a closing of a given X server client. This may not necessarily close the actual misbehaving process, but most graphical programs will terminate immediately if the X server disconnects them.

```
cmdKill :: WMCommand ()
cmdKill = liftWM $ withDisplay $ λdpy → do
  fw ← focusWindow
  case fw of
    Just mw → do liftIO $ killClient dpy $ window mw
      return ()
    Nothing → return ()
```

```
cmdTest :: WMCommand ()
cmdTest = liftWM $ recurse (50, 50, 1, 0)
  where recurse :: (Integer, Integer, Integer, Integer) → WM ()
    recurse (_, _, _, 500) = return ()
    recurse (99, x2, 1, c) = recurse (99, x2, −1, c)
    recurse (x1, 99, −1, c) = recurse (x1, 99, 1, c)
    recurse (x1, x2, v, c) = do
      ff ← gets focusFrame
      changeFrames $ λ(t, f) →
        (changeAtPath t (init ff) $ λs →
          case s of
            (Split dir (_, f1) (_, f2)) →
              Split dir (x1, f1) (x2, f2)
            _ → s
        , f)
      liftIO $ threadDelay 1
      recurse (x1 + v, x2 − v, v, c + 1)

cmdExec :: WMCommand ()
cmdExec = do
  s ← accept $ String "/bin/sh -c "
  liftWM $ spawnChild s $ λstatus →
    case status of
      Exited (ExitFailure code) →
        message $ "/bin/sh -c \""
          ++ s
          ++ "\" finished ("
          ++ show code
          ++ ")"
      _ → return ()
```

The colon command reads in a single string parameter and evaluates it as a Mousetoxin command string. The name is a reference to the fact that the default keybinding is the colon. For ease of use we would like to support tab-completion for the first part of the parameter, namely the command name; the arguments taken by a given command are unfortunately opaque, so we cannot complete those. To implement this, we start by defining a presentation type and a *CommandArg* instance (see Section 3.8 on page 34).

```
data CommandString = CommandString String
instance CommandArg CommandString String where
  accept (CommandString pstr) = do
    cmds ← liftWM (M.keys <$> commands <$> asks config)
    let compl str = filter (isPrefixOf $ cmdpart str) cmds
      finish str = if elem (cmdpart str) cmds then
        Right str else
        Left $ "Unknown command '" ++ str ++ "'."
    accepting pstr (return ∘ compl) (return ∘ finish)
      where cmdpart = takeWhile (≢ ' ')
```

We can now define the colon command.

```
cmdColon :: WMCommand ()
cmdColon = do s ← accept $ CommandString ":"
  liftWM $ evalCmdString s

cmdHoriSplit :: WMCommand ()
cmdHoriSplit = liftWM $ splitFocus Horizontal Left
cmdVertSplit :: WMCommand ()
cmdVertSplit = liftWM $ splitFocus Vertical Left

cmdUnsplitAll :: WMCommand ()
cmdUnsplitAll = do
  liftWM $ do
    fw ← focusWindow
    changeFrames $ λ(t, _) →
      (changeAtPath t [] $ (const (Frame $ fw ≫= return ∘ window))
      , [])

cmdNextFrame :: WMCommand ()
cmdNextFrame = liftWM $ do
  ps ← leafPaths < $ > gets frames
  curr ← gets focusFrame
  case dropWhile (≢ curr) $ ps ++ ps of
    (_ : new : _) → focusOnFrame new
    _ → return ()

cmdResize :: WMCommand ()
cmdResize = liftWM $ withDisplay $ λdpy → do
  ff ← gets focusFrame
  allmanaged ← M.toList < $ > gets managed
  when (¬ $ null ff) $ withGrabbedKeyboard $ do
    let screen = defaultScreenOfDisplay dpy
      managedWindow win = snd < $ > find ((≡) win ∘ window ∘ snd) allmanaged
      change d = changeFrames $ λ(t, f) →
        (changeAtPath t (init ff) $
          increaseBy d screen managedWindow
        , f)
      keymap = [((controlMask, xK_n), change 1 ≫ loop)
        , ((controlMask, xK_p), change (−1) ≫ loop)
        , ((controlMask, xK_f), change 1         ≫ loop)
        , ((controlMask, xK_b), change (−1) ≫ loop)
        , ((controlMask, xK_g), return ())
        , ((controlMask, xK_Escape), return ())]
      loop = do ((s, m), _) ← readKey
               fromMaybe loop $ lookup (m, s) keymap
    loop
    where increaseBy d screen manwin (Split dir (x1, f1) (x2, f2)) =
           Split dir (split′, f1)
```

```
        (dim - split', f2)
      where split = fromIntegral x1 / fromIntegral (x1 + x2) :: Double
            dim   = fromIntegral $
                      case dir of
                        Horizontal → heightOfScreen screen
                        Vertical → widthOfScreen screen
            split' = bound 1 (dim - 1)
                      (truncate (split * fromIntegral dim)
                        + d * lcm (incunit f1 dir manwin)
                          (incunit f2 dir manwin))
    increaseBy _ _ _ ft = ft    -- Should never happen.
    incunit (Frame Nothing) _ _ = 1
    incunit (Frame win) dir manwin = fromMaybe 1 $ do
      (w, h) ← sh_resize_inc =<< sizeHints <$> (manwin =<< win)
      case dir of
        Horizontal → return $ fromIntegral h
        Vertical   → return $ fromIntegral w
    incunit (Split _ (_, f1) (_, f2)) dir manwin =
        lcm (incunit f1 dir manwin)
          (incunit f2 dir manwin)
    bound lower upper = max lower ∘ min upper
```

The *banish* command exiles the mouse pointer to a disgraceful position in the lower right of the screen, outside the users working area. We actually have to put it slightly offset from the actual lower right, or it will seemingly wrap around to the upper left.

```
cmdBanish :: WMCommand ()
cmdBanish = liftWM $ withDisplay $ λdpy → do
  let scr = defaultScreenOfDisplay dpy
  rootw ← asks rootW
  liftIO $ warpPointer dpy none rootw 0 0 0 0
    (fromIntegral $ widthOfScreen scr - 2)
    (fromIntegral $ heightOfScreen scr - 2)
```

The *meta* command sends the prefix key to the focus window (if any). This is needed for applications to receive the prefix key at all (since normal keyboard input will be caught by Mousetoxin). Due to a gregarious flaw in the Xlib binding, we have to manually set the type field of our synthetic event via a low level byte-manipulation function. The byte order of the structure we manipulate is defined in terms of a C struct in the Xlib documentation, so it should be safe, just not very pretty.

```
cmdMeta :: WMCommand ()
cmdMeta = liftWM $ withDisplay $ λdpy → do
  focus ← focusWindow
  let send fw = do
    (km, ks) ← prefixKey <$> asks config
    rootw ← asks rootW
    liftIO $ allocaXEvent $ λev → do
```

```
        kc ← keysymToKeycode dpy ks
        setKeyEvent ev (window fw) rootw none km kc True
        pokeByteOff ev 0 keyPress
        sendEvent dpy (window fw) False keyPressMask ev
        sync dpy False
  maybe (return ()) send focus
```

```
cmdQuitMousetoxin :: WMCommand ()
cmdQuitMousetoxin = liftIO exitSuccess
```

```
cmdTime :: WMCommand ()
cmdTime = do timezone ← liftIO getCurrentTimeZone
  timeUTC ← liftIO getCurrentTime
  let localTime = utcToLocalTime timezone timeUTC
  liftWM $ message $ formatTime defaultTimeLocale "%a %b %d %H:%M:%S %Y" localTime
```

A command that aborts the current invocation attempt is occasionally useful, for example for dismissing the overlay window.

```
cmdAbort :: WMCommand ()
cmdAbort = return ()
```

The *newwm* command starts another given window manager in place of Mousetoxin. The primary complexity is disabling enough of our dynamic environment that a new window manager can start up properly, yet leave it sufficiently intact to restore ourselves if the other window manager cannot be executed. In practice, we must take the following steps:

- Unmap the overlay window so that the new window manager will not try to take control of it
- Map all managed windows so the new window manager can find and control them.
- Relinquish our exclusive grab on *SubstructureRedirectMask* events on the root window.

We don't need to keep track of exactly which windows were already mapped if we need to restore the original configuration, as we assume that *changeFrames* will ensure that the window managers notion of visible windows corresponds to which windows are actually mapped in the X server.

```
cmdNewWM :: WMCommand ()
cmdNewWM = do newwm ← accept $ String "Switch to wm:"
  liftWM $ withDisplay $ λdpy → do
    root ← asks rootW
    overlay ← asks overlayWindow
    liftIO $ do unmapWindow dpy overlay
      selectInput dpy root 0
    wins ← map (window ∘ snd) <$> M.toList <$> gets managed
```

```
    liftIO $ do forM_ wins $ mapWindow dpy
      sync dpy False
      selectInput dpy root 0
      executeFile newwm True [] Nothing
        `Prelude.catch` (const $ return ())
        -- Couldn't execute, restore...
      selectInput dpy root rootMask
      forM_ wins $ mapWindow dpy
    changeFrames id
    message $ "Could not execute '" ++ newwm ++ "'."
```

While replacing the running Mousetoxin instance with another program (as in *newwm*) is fairly simple to do properly, the *tmpwm* command, which temporarily relinquishes window management control to another process, is impossible to implement without quirks in the general case.

```
cmdTmpWM :: WMCommand ()
cmdTmpWM = do newwm <- accept $ String "Tmp wm:"
  liftWM $ withDisplay $ \dpy -> do
    root <- asks rootW
    overlay <- asks overlayWindow
    wins <- map (window . snd) <$> M.toList <$> gets managed
    status <- liftIO $ do
      unmapWindow dpy overlay
      selectInput dpy root 0
      ungrabKey dpy anyKey anyModifier root
      sequence ([mapWindow dpy,
        flip (selectInput dpy) 0,
        ungrabKey dpy anyKey anyModifier] <*> wins)
      sync dpy False
      selectInput dpy root 0
      pid <- forkProcess $ do
        uninstallSignalHandlers
        executeFile newwm True [] Nothing
          `Prelude.catch` (const $ return ())
        -- Wait for the subproc to end, then restore,
        -- returning status
      getProcessStatus True False pid
        <* mapWindow dpy overlay
        <* selectInput dpy root rootMask
    grabKeys root
    maybe (restore wins >>
      message ("Could not execute '" ++ newwm ++ "'."))
          (const rescan) status

where rescan = do modify $ \s -> s
  { managed = M.empty
  , frames     = Frame Nothing
  , focusFrame = []
  }
```

```
scanWindows
message "Mousetoxin is back!"
restore wins = withDisplay $ λdpy → do
  sequence ([liftIO ∘ flip (selectInput dpy) clientMask,
    grabKeys] <*> wins)
  changeFrames id
```

Chapter 6

Mousetoxin.InputEditor

In this chapter we will implement the standard commands for input editing in Mousetoxin. The self-insertion command, by which most text is actually entered, is part of the editor code itself, and is thus found in Chapter 3 on page 5, while the bindings from keyboard invocations to commands if part of the configuration described in Chapter 7 on page 59. We shall prefix all exported commands with *edit* for clarity, as in *editBackChar*.

```
module Mousetoxin.InputEditor (editAbort
  , editDone
  , editBackwardChar
  , editForwardChar
  , editBackwardWord
  , editForwardWord
  , editBeginningOfLine
  , editEndOfLine
  , editForwardDeleteChar
  , editBackwardDeleteChar
  , editCompleteNext
  , editCompletePrev
  ) where
import Control.Applicative
import Data.Char
import Data.List
import Data.Maybe
import Control.Monad.State
import Control.Monad.Reader
import Mousetoxin.Core
```

The abort command merely returns the *Abort CommandResult*. Recall that this will cause input editing to terminate abnormally.

```
editAbort :: EditCommand CommandResult
editAbort = return Abort
```

A similar case is the finishing command, which will cause input editing to terminate normally.

```
editDone :: EditCommand CommandResult
editDone = return Done
```

Many commands will do nothing but change the editing point by some measure. To help with this, we define a helper function that will return *Fail* if it cannot move at all, *No_Op* if it can move at least a single character.

```
pointBounds :: Int → EditCommand (Int, String)
pointBounds d = do
  buf ← gets inputBuffer
  if (d < 0)
    then return (0, "Beginning of buffer")
    else return (length buf, "End of buffer")
movePoint :: Int → EditCommand CommandResult
movePoint d = do ep ← gets editingPoint
                 (bound, emsg) ← pointBounds d
                 if (ep ≡ bound)
                   then return $ Fail emsg
                   else do modify $ λs → s{ editingPoint = ep + d}
                     return No_Op
```

The two most basic commands move the editing point forward and backward.

```
editBackwardChar :: EditCommand CommandResult
editBackwardChar = movePoint (−1)
editForwardChar :: EditCommand CommandResult
editForwardChar = movePoint 1
```

Two useful commands move by words, which we (and Ratpoison) interpret to mean across all non-alphanumerics in sequence, then across all alphanumerics in sequence. They are both defined in terms of more primitive functions that move point until the character at point fulfils some given predicate.

```
moveBackwardUntil :: (Char → Bool) → EditCommand CommandResult
moveBackwardUntil p = do
  point ← gets editingPoint
  buf ← gets inputBuffer
  let end = fromMaybe point (findIndex p $ reverse $ take point buf)
    d = −end
  movePoint d
editBackwardWord :: EditCommand CommandResult
editBackwardWord = do moveBackwardUntil isAlphaNum
  moveBackwardUntil (¬ ∘ isAlphaNum)
moveForwardUntil :: (Char → Bool) → EditCommand CommandResult
moveForwardUntil p = do
  point ← gets editingPoint
  buf ← gets inputBuffer
  let end = fromMaybe (length buf − point) (findIndex p $ drop point buf)
    d = end
  movePoint d
```

```
editForwardWord :: EditCommand CommandResult
editForwardWord = do moveForwardUntil isAlphaNum
  moveForwardUntil (¬ ∘ isAlphaNum)
```

As the input editor is single-line, the commands for moving point to the beginning and end of line are quite simple indeed, merely moving point to zero and the size of the input buffer respectively.

```
editBeginningOfLine :: EditCommand CommandResult
editBeginningOfLine = do modify $ λs → s{ editingPoint = 0}
  return No_Op
editEndOfLine :: EditCommand CommandResult
editEndOfLine = do modify $ λs → s{ editingPoint = length $ inputBuffer s}
  return No_Op
```

To support our selection of deletion commands, we capture common functionality by defining a simple helper function to delete a *range* of characters in the input buffer. The function returns the deleted string, not a *CommandResult*, as we do not consider a deletion to be a failure even if there are no characters to delete.

```
deleteRange :: Int → Int → EditCommand String
deleteRange x y = do
  before ← take low < $ > gets inputBuffer
  str    ← take diff < $ > drop low < $ > gets inputBuffer
  after ← drop high < $ > gets inputBuffer
  modify $ λs → s{ inputBuffer = before ++ after}
  return str
    where low = min x y
      high = max x y
      diff = high − low
```

The backward and forward deleting functions (*delete* and *backspace* to most users) are defined to simply delete one-character ranges.

```
editForwardDeleteChar :: EditCommand CommandResult
editForwardDeleteChar = do point ← gets editingPoint
  (bounds, msg) ← pointBounds 1
  if (point ≡ bounds)
    then return $ Fail msg
    else do deleteRange point $ point + 1
      return No_Op
editBackwardDeleteChar :: EditCommand CommandResult
editBackwardDeleteChar = do point ← gets editingPoint
  deleteRange point $ point − 1
  editBackwardChar
```

Completion is somewhat involved due to the heavy reliance on editor state. At the basic level, we wish to replace the text to the left of the editing point with some completion of that text. However, if there are multiple valid completions, the user must be provided with a way to cycle though them. This is the purpose of the *completing* field in the editor state, a field that is either *Nothing* (indicating that we are not currently cycling through completions), or a list of completions tagged with *Just*. In this way, we can support completion-cycling through multiple command invocations. The alternative would be for the completion command to read keypress events from the X server on its own, and cycle through completions as long as it receives Tab keypresses (or whatever is appropriate). This, however, would be very messy and require duplicating a large amount of complex input logic within the completion command.

As a lesser consideration, we also do not impose a strict way of cycling through the completion list. We intend to support at least forwards and backwards cycling, and we might as well permit a general selection function.

```
completeByDirec :: (String → [String] → Maybe String)
   → EditCommand CommandResult
completeByDirec select = do
  prepoint  ← liftM2 take (gets editingPoint) (gets inputBuffer)
  postpoint ← liftM2 drop (gets editingPoint) (gets inputBuffer)
  let cycleCompletions l =
          case select prepoint l of
            Nothing → return No_Op
            Just c → do
              modify $ λs → s{ inputBuffer = c ++ postpoint
                 , editingPoint = length c}
              return Completion
    newCompletions = do
      c ← asks completer
      compls ← liftWM $ c prepoint
      case compls of
        [] → return No_Op
        (comp : rest) → do
          modify $ λs → s{ inputBuffer = comp ++ postpoint
             , editingPoint = length comp
             , completing = Just $ comp : rest}
          return Completion
  compl    ← gets completing
  case compl of
    Just [] → return No_Op
    Just l  → cycleCompletions l
    Nothing → newCompletions
```

The concrete commands for completion are now fairly simple. The command that moves forward through the completion list, in case we are cycling, merely identifies the location of the current completion, and yields the next one. Doubling the list of completions is a clever trick to ensure that we'll find a "next" completion even if the current completion is last in the list.

```
editCompleteNext :: EditCommand CommandResult
editCompleteNext = completeByDirec $ λs l →
  case dropWhile (≢ s) (l ++ l) of
    (_ : c : _) → Just c
    _ → Nothing
```

The command for backwards cycling is just as simple, merely reversing the list before finding the "next" completion.

$$
\begin{array}{l}
\mathit{editCompletePrev} :: \mathit{EditCommand}\ \mathit{CommandResult} \\
\mathit{editCompletePrev} = \mathit{completeByDirec}\ \$\ \lambda s\ l \rightarrow \\
\quad \mathbf{case}\ \mathit{dropWhile}\ (\not\equiv s)\ (\mathit{reverse}\ \$\ l +\!\!+\ l)\ \mathbf{of} \\
\qquad (\_ : c : \_) \rightarrow \mathit{Just}\ c \\
\qquad \_ \rightarrow \mathit{Nothing}
\end{array}
$$

Chapter 7

Mousetoxin.Config

This chapter describes the default user configuration of Mousetoxin (represented by the *WMConfig*–type). We also implement a facility for parsing a user configuration file. The actual commands are defined in Chapter 5 on page 45.

```haskell
module Mousetoxin.Config
  (versionString
  , defaultConfig
  ) where
import Mousetoxin.Core
import Mousetoxin.Commands
import Mousetoxin.InputEditor
import Graphics.X11.Types
import Data.Map as M
```

Mousetoxin follows a very simple versioning scheme, where development versions are indicated by suffixing the string `-dev`.

```haskell
versionString :: String
versionString = "1.0-dev"
```

```haskell
defaultConfig :: WMConfig
defaultConfig = WMConfig
  { displayStr = ":0.0"
  , prefixKey  = (controlMask, xK_t)
  , overlayBorderWidth = 1
  , overlayPadding = (4, 0)
  , keyBindings = M.fromList [((0, xK_n), "next")
                 , ((0, xK_p), "prev")
                 , ((0, xK_0), "select 0")
                 , ((0, xK_1), "select 1")
                 , ((0, xK_2), "select 2")
                 , ((0, xK_3), "select 3")
                 , ((0, xK_4), "select 4")
                 , ((0, xK_5), "select 5")
                 , ((0, xK_6), "select 6")
```

```
                     , ((0, xK_7), "select 7")
                     , ((0, xK_8), "select 8")
                     , ((0, xK_9), "select 9")
                     , ((controlMask, xK_t), "other")
                     , ((0, xK_a), "time")
                     , ((0, xK_w), "windows")
                     , ((0, xK_g), "abort")
                     , ((0, xK_k), "close")
                     , ((0, xK_K), "kill")
                     , ((0, xK_c), "exec urxvt")
                     , ((0, xK_exclam), "exec")
                     , ((0, xK_colon), "colon")
                     , ((0, xK_s), "hsplit")
                     , ((0, xK_S), "vsplit")
                     , ((0, xK_Q), "unsplitall")
                     , ((0, xK_Tab), "nextframe")
                     , ((0, xK_r), "resize")
                     , ((0, xK_b), "banish")
                     , ((0, xK_t), "meta")
                     , ((0, xK_y), "test")
                     , ((mod1Mask, xK_q), "quit")
                     , ((controlMask, xK_g), "abort")]
, commands = M.fromList [("next", cmdNext)
                     , ("prev", cmdPrev)
                     , ("select", cmdSelect)
                     , ("other", cmdOther)
                     , ("time", cmdTime)
                     , ("windows", cmdWindows)
                     , ("delete", cmdClose)
                     , ("close", cmdClose)
                     , ("kill", cmdKill)
                     , ("exec", cmdExec)
                     , ("colon", cmdColon)
                     , ("hsplit", cmdHoriSplit)
                     , ("vsplit", cmdVertSplit)
                     , ("unsplitall", cmdUnsplitAll)
                     , ("only", cmdUnsplitAll)
                     , ("nextframe", cmdNextFrame)
                     , ("focus", cmdNextFrame)
                     , ("resize", cmdResize)
                     , ("banish", cmdBanish)
                     , ("meta", cmdMeta)
                     , ("test", cmdTest)
                     , ("quit", cmdQuitMousetoxin)
                     , ("abort", cmdAbort)
                     , ("newwm", cmdNewWM)
                     , ("tmpwm", cmdTmpWM)]
, editCommands = M.fromList
    [((controlMask, xK_g), editAbort)
    , ((0, xK_Escape), editAbort)
```

$, ((0, xK\_Return), editDone)$
$, ((controlMask, xK\_b), editBackwardChar)$
$, ((0, xK\_Left), editBackwardChar)$
$, ((controlMask, xK\_f), editForwardChar)$
$, ((0, xK\_Right), editForwardChar)$
$, ((mod1Mask, xK\_b), editBackwardWord)$
$, ((mod1Mask, xK\_f), editForwardWord)$
$, ((controlMask, xK\_a), editBeginningOfLine)$
$, ((0, xK\_Home), editBeginningOfLine)$
$, ((controlMask, xK\_e), editEndOfLine)$
$, ((0, xK\_End), editEndOfLine)$
$, ((controlMask, xK\_d), editForwardDeleteChar)$
$, ((0, xK\_Delete), editForwardDeleteChar)$
$, ((0, xK\_BackSpace), editBackwardDeleteChar)$
$, ((0, xK\_Tab), editCompleteNext)$
$, ((0, 0\ xfe20), editCompletePrev)]$
$\}$