Implementing Real-time Interactive Multi-User Games with Fudgets

Magnus Carlsson & Thomas Hallgren

{magnus,hallgren}@cs.chalmers.se

1 Introduction

This paper consists of two parts: the first part (section 2) illustrates how the Fudget system [1][2] can be used to write real-time interactive games in Haskell. This shows that the Fudget system is not limited to traditional, fairly static, graphical user interfaces, but also allows you to construct interfaces with lots of animated objects. A program with one concurrent process (one fudget) per animated object will have a much nicer structure than a single-threaded, sequential program.

The second part of the paper (section 3) describes a way to extend the Haskell I/O system to make it possible to write multi-user games. We introduce mechanisms for communication over the network with other processes (by means of UNIX sockets) and for making indeterministic choices. This is useful for client/server programming in general; multi-user games is just one particular example. section 3 gives an example of a client/server application called Chat, where the server distributes messages from connected clients to all clients.

2 Interactive Real-Time Games

The Fudget system is implemented on top of the X windows system. A Fudget program is essentially a function from a stream of X events to a stream of X commands (see Figure 1). This makes it possible to write interactive programs with graphical user interfaces. A program is either in a resting state waiting for some X event, or busy computing some X commands as a response to an X event. Most X events are caused by some keyboard or mouse activity.

The X windows interface is implemented by an extension to Haskell's stream based I/O system. but in the following sections it is sufficient to think of a Fudget program as a function of type

```
[XEvent] → [XCommand]
```

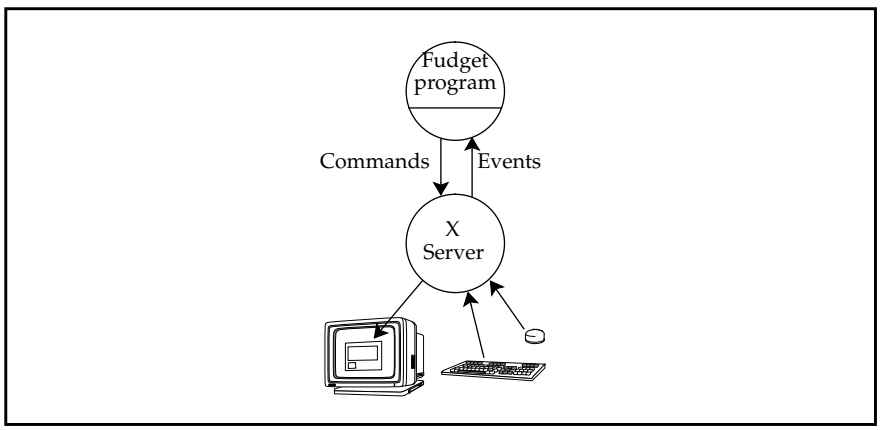

Figure 1: A Fudget program's connection with the outside world

where the type `XCommand` contain constructors corresponding to various calls to the Xlib library [5] and `XEvent` contain constructors corresponding to the various X events.

2.1 Real-time extension to the X Windows interface

In real-time programs, output is affected not only by the flow of input from the user, but also by the flow of time in the real world. Output is produced not only as response to user input, but also at specific moments in time. To make it possible to write Fudget programs with real-time behaviour we have made the following extensions to the `XCommand` and `XEvent` types[1]:

```
data XCommand =
    ...
    | SetTimer Int Int Int      -- first delay, interval, timer identity
    | RemoveTimer Int           -- timer identity

data XEvent =
    ...
    | TimerAlarm Int            -- timer identity
```

After outputting the command `SetTimer` *d i n*, a program will receive `TimerAlarm` *n* in the input `XEvent` stream at approximately the times *d+k*i* milliseconds after the `SetTimer` command was output, where *k* is a non-negative integer. The timer alarms are merged in chronological order with the ordinary `XEvent` input, saving us from having to introduce some explicit indeterministic merge operation in the language. This technique is known as hiatonic input [3].

2.2 Space Invaders

In this section we describe an implementation of the classical game *Space Invaders*. Only the most fundamental parts of the game has actually been implemented (see Figure 2),

In this game, an army of invaders from outer space is approaching the earth. The player must shoot them all down before they reach the surface. The player controls a gun, which can be moved horizontally at the bottom of the screen (the surface of the earth) and which can fire vertically. The invaders initially move from left to right. When the right most invader reaches the right edge of the screen all

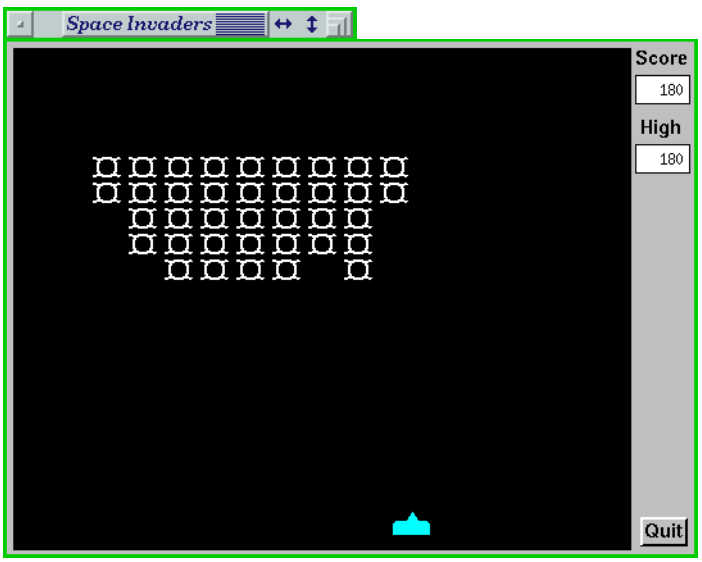

Figure 2: Space Invaders - a typical interactive real-time game

1. This extension was designed and implemented by Lennart Augustsson.

invaders first move downwards a small distance, then move horizontally again until the left most invader reaches the left edge, and so on.

2.2.1 Structure of the Space Invaders implementation

In this section we describe an implementation of Space Invaders, where the each object is implemented as a fudget. The objects are:

1. `spaceF`: the space fudget. This is the black background in which all the other objects move around.
2. `gunF`: the gun.
3. `torpedoF`: the torpedoes fired by the gun.
4. `invaderF`: a number of invaders

`gunF` and `torpedoF` use timers internally to control the speed of their motion. To coordinate the motion of the invaders, they are controlled by a common timer which is located in a windowless fudget called `timerF`. There is also an abstract fudget called `shoutSP`, which broadcasts timer alarms and other input to all invaders.

Figure 3 illustrates how the fudgets are interconnected. The information flow is as follows: the space fudget outputs mouse and keyboard events to `gunF`. (This allows the user to place the mouse pointer anywhere in the window to control the gun.) The gun respond to these events by starting or stopping to move, or by firing a torpedo. When the gun is fired, it outputs its current position to the torpedo fudget. The torpedo then starts moving upwards from that position. When it hits something, it outputs its current position to the invaders. Each invader then checks if the hit is within the area it occupies on the screen and, if so, it removes its window and dies.

Below, we take a closer look at `invaderF`. The other fudgets are just variations on a theme, so we won't discuss them further.

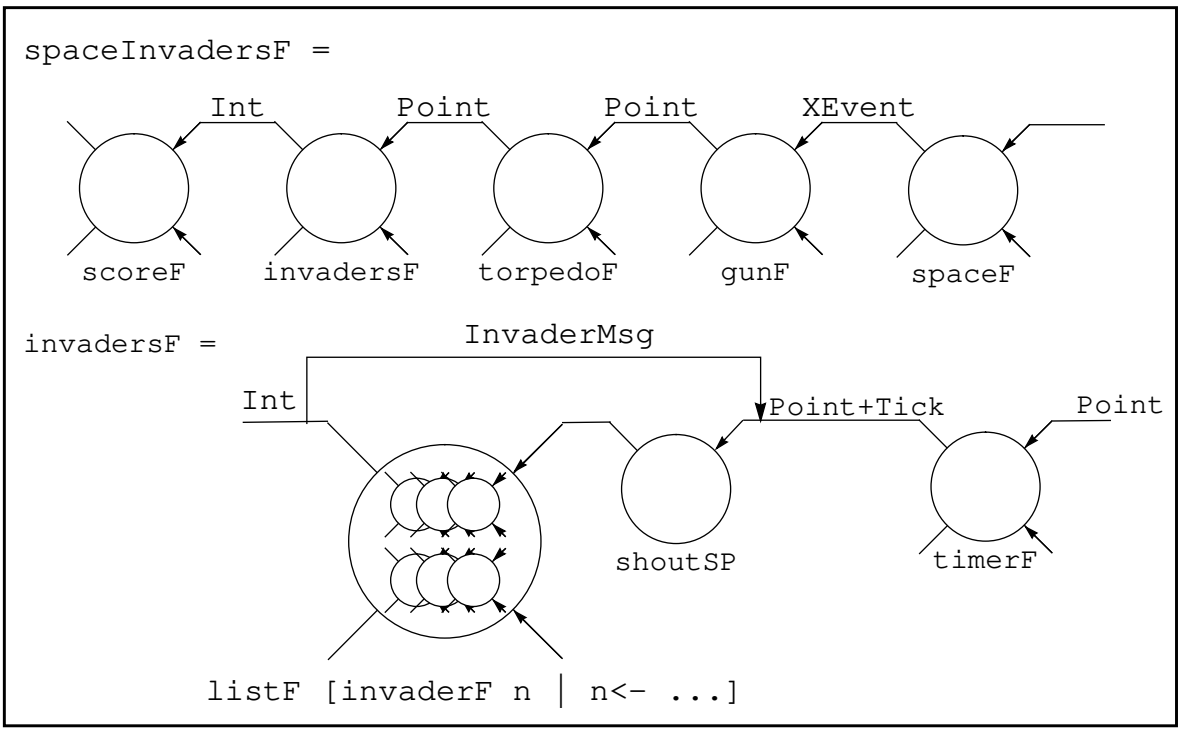

Figure 3: The processes and their interconnection in the Space Invaders implementation.

`invaderF` maintains an internal state consisting of the following parts: the current position (a `Point`), the current direction (left or right), if it is time to turn (i.e., move downward at the next timer alarm, and then change directions).

The invaders speak the following language:

```
data InvaderMsg = Tick | Turn | Hit Point | Death (Int,Int)
```

When an invaders hears a `Tick`, it moves one step in the current direction. It also checks if it has reached an edge, in which case it outputs `Turn`, which is received by all invaders. When an invader hears a `Turn` it remembers that it is time to turn at the next `Tick`. When a torpedo has hit something at position *p*, all invaders receive `Hit` *p*, and check if *p* is within their screen area. If so, it outputs `Death` *n*, where *n* is the identity of the invader. *n* is used by `shoutSP`, so that it doesn't have to shout to dead invaders. It is also used to determine how many points to add to the score.

The fact that all objects are implemented as fudgets mean that each object has its own X window. To move an object you move its window. No drawing commands need to be output.

How does the torpedo know if it has hit something? The torpedo is a window which moves behind all other windows. This means that it becomes obscured when it hits something. The X server sends a `VisibilityNotifty` event when this happens. This causes the torpedo to stop and send its current position to the invaders.

2.2.2 About the efficiency of the Space Invaders implementation

One major point of the Fudget system (and of functional programming in general) is to simplify and speed up program development. But it is of course also important that the efficiency of the resulting program is acceptable.

We have measured the CPU time consumption of the Space Invaders implementation described above running on a Sparcstation IPX in a situation where 55 invaders move twice per second, the gun and the torpedo move every 30ms. The average CPU load was approximately 60%. 10% of this was consumed by the X server. As a comparison, the program xinvaders, a C program implemented directly on top of Xlib, consumes less than 5% CPU time in a similar situation.

As usual, programming on a higher abstraction level results in a less efficient solution. Part of the inefficiency comes from the use of Haskell and the Fudget system. The load on the X server comes from the fact that the moving objects are represented as windows. Not surprisingly, moving a window is a more expensive operation than just drawing an image of the same size. But using techniques outlined in the next section, it is possible to rewrite the Fudget program to draw in a single window, like the C program, and still keep the same nice program structure, i.e., one process per moving object.

Above, we compared the efficiency of a high level implementation (using the Fudget system) of the game with a low level implementation. It would also be interesting compare other user interface toolkits, e.g. Motif and Interviews, to the Fudget system.

The CPU time consumption figures above does not say much about the real-time behaviour of the two implementations. The fact is that the C program meets the real-time deadlines, but the Fudget program does not. As a response to a `Tick` from `timerF`, all 55 invaders should move one step. Computing and outputting 55 `MoveWindow` commands unfortunately takes much more than 30ms, which means that the `MoveWindow` commands for the gun and the torpedo will be output too late, resulting in a very jerky motion. This problem can be solved in at least two different ways: manually, by not moving all 55 invaders at the same time and thus not blocking output from other fudgets for longer than 30ms; automatically (from the point of view of the application programmer), by introducing parallel evaluation and some kind of fair, indeterministic merge of the output from different fudgets. The latter solution is of course the more general one, and we hope to improve the Fudget system in this direction.

2.3 Programming with concurrent processes in Haskell

Above, we outlined a program structure where each moving object on the screen is represented as a process (a Fudget). Each process controls a window on the screen. It is of course possible to generalise this and use processes for purposes other than controlling user interface elements.

Processes in the Fudget library are called *stream processors* and are represented by the type `SP a b`, where `a` is the type of input messages and `b` is the type of output messages. Stream processors are programmed in a continuation style using the following three basic constructors:

```
nullSP :: SP a b                              -- does nothing
putSP  :: [b] → SP a b → SP a b               -- writes to the output stream
getSP  :: (a→SP a b) → SP a b                 -- reads from the input stream
```

The behaviour of a single Fudget is usually implemented as one sequential program by using these operators. Then there are combinators for parallel and serial composition of stream processors, on which the corresponding Fudget combinators are based.

```
serCompSP :: SP a b → SP c a → SP c b
compSumSP :: SP a b → SP c d → SP (a+c) (b+d)
```

Input to a parallel composition, `sp1 `compSumSP` sp2`, is delivered to one of `sp1` and `sp2`. Sometimes it is more natural to broadcast the input to all processes in a parallel composition. Recall from section 2.2.1 that we used a separate stream processor `shoutSP` for this purpose. Some overhead can be avoided by using a tailor made combinator for parallel composition with broadcast instead. We therefore introduce `parSP`:

```
parSP :: SP a b → SP a b → SP a b
```

This also makes it easy to write stream processors that dynamically split into two or more parallel processes. One of the processes in a parallel composition can terminate without leaving any overhead behind.

```
nullSP `parSP` sp == sp `parSP` nullSP == sp
```

We can also introduce a sequential composition operator:

```
seqSP :: SP a b → SP a b → SP a b
```

`sp1 `seqSP` sp2` behaves like `sp1` until `sp1` becomes `nullSP`, and then behaves like `sp2` (without any overhead).

These new operators provide a more flexible way to program the behaviour of single fudgets. For example they can be used in the Space Invaders program to keep the structure with one process per moving object although all drawing is done in one window.

However, the stream processor idea presented in this section, although designed in the context of the Fudget system, is independent of the Fudget system and can surely be useful when solving other programming problems which can be decomposed as a number of concurrent processes.

3 Multi-User Games

3.1 Input/Output in Haskell

The Haskell Report [6] describes input/output in Haskell in terms of streams. Today, monadic I/O has become increasingly popular [7], but the original presentation with the data types `Request` and `Response` fits better with the stream processor model used in Fudgets. Therefore, the necessary extensions will be made in the stream-I/O model.

3.1.1 Dialogue, Response and Request

A Haskell program has the type `Dialogue`, where

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

From the view of the operating system, the program is a stream processor that emits requests, and is to be fed with the responses corresponding to these requests. The data types `Request` and `Response` are:

```
data Request =
   -- file system requests:
      ReadFile String
    | WriteFile String String
    | AppendFile String String
    | DeleteFile String
    | StatusFile String
   -- channel system requests:
    | ReadChan String
    | AppendChan String String
    | StatusChan String
   -- environment requests:
    | Echo Bool
    | GetArgs
    | GetEnv String
    | SetEnv String String

data Response =
      Success
    | Str String
    | StrList [String]
    | Bn Bin
    | Failure IOError
```

The `Request` data type has requests regarding the file system, channel system, and the environment. In client/server applications, the communication between programs is carried out on channels. Unfortunately, the channel system as defined in Haskell is too weak for our purposes. There is only a fixed set of channels, namely `stdin`, `stdout`, `stderr`, and `stdecho`. We need the possibility to dynamically create new channels to clients via Unix sockets. Therefore, we suggest the additional requests and responses

```
data Request =
    ...                                        -- expected response:
    | OpenLSocket LSocketAddress               -- GotLSocket
    | OpenSocket SocketAddress                 -- GotSocket
    | CloseLSocket LSocket                     -- Success
    | CloseSocket Socket                       -- Success
    | AcceptSocket LSocket                     -- GotSocket
    | ReadSocket Socket                        -- Str
    | WriteSocket Socket String                -- Success

data Response =
    ...
    | GotLSocket LSocket
    | GotSocket Socket

type LSocketAddress = String
type SocketAddress = String

data LSocket
data Socket
```

The requests are similar to the existing channel system requests, but instead of using strings for encoding channels, we use two primitive handle types, `LSocket` and `Socket`. `LSocket` corresponds to Unix sockets that are used to accept connections (Listener sockets), and `Socket` is a channel which can be read from or written to (corresponds to a connected or accepted Unix socket).

The requests `AcceptSocket` and `ReadSocket` (and also `ReadChannel`) turn out to be rather useless in a setting where we may expect input from several sources, without knowing which of them will come first. Therefore, we need a mechanism for merging input from many channels non-deterministically. In Appendix D of the Haskell report, optional requests and responses are specified for this purpose, namely `ReadChannels` and `Tag`:

```
data Request  = ... | ReadChannels [String]
data Response = ... | Tag [(String, Char)]
```

The `ReadChannels` request takes a list of channel names (strings) as argument, and the response is `Tag` with a merged and tagged list of characters from the channels. The channel names are used as tags.

To handle new connections and incoming data on sockets, we add the requests

```
data Request =
    ...                              -- expected response:
    | AcceptSockets [LSocket]        -- Success
    | ReadSockets [Socket]           -- Success
```

Since the set of channels that a program wants to watch may vary, it will typically request `AcceptSockets` and `ReadSockets` repeatedly. However, we only want a single list of tagged channel data to take care of. Actually, we only want one single list for all asynchronous input to the program, including channel data, events from the X-server, and timer alarms. This is the reason why `AcceptSockets` and `ReadSockets` do not return tagged lists, as ReadChannels did. To get hold of *the* asynchronous input list, we add

```
data Request = ... | GetAsyncInput

data Response = ... | GotAsyncInput [AsyncInput]

data AsyncInput =
      AISocketAccepted LSocket Socket
    | AISocketRead Socket String
    | AITimerAlarm Timer
    | AIXEvent XEvent
```

If a listener socket `ls` is included in an `AcceptSockets` request, and someone connects to `ls`, the value `AISocketAccepted ls s` will be appended to the asynchronous input list by the run-time system. Here, `s` is a socket which will be used for communication to the new connection. Then, if the program emits `ReadSockets [s]`, input on the connection will result in values like `AISocketRead s data`, where `data` are strings received.

3.2 Example: Chat

In this section, we will present a small sample application using the socket facilities. It is called Chat, and allows many people to connect to a server and send messages to each other. Any message that a user enters on his/her client, will be broadcast to all clients currently connected. Clients may connect

and disconnect at any time. The other clients are notified when such events occur. In Figure 4, a screen dump shows a typical Chat session from one client's point of view.

The Chat application consists of two Haskell programs, the server and the client.

3.2.1 The server program

The server employs the fudget `socketServerF`, of type:

```
socketServerF :: LSocketAddress -> (Socket -> F a (SocketMsg b))
                 -> F (Int,a) (Int, ServerMsg b)

data SocketMsg a = SocketMsg a | SocketEOS
data ServerMsg a = ServerMsg a | ServerEOS | ServerNew
```

The first argument is a listener socket address, which `socketServerF` will open and listen to. The second argument is a client handler fudget. Whenever a new connection is accepted, `socketServerF` will launch a new client handler fudget with the new socket as argument. It will also emit the message `(i,ServerNew)`, where `i` is an integer tag used for communication with the handler. The handler can then emit messages, for example, `SocketMsg "Hello"`. This will be tagged and emitted by `socketServerF` as `(i,ServerMsg "Hello")`. If the handler wants to terminate, it emits `SocketEOS`. This results in the handler fudget being destroyed, and the message `(i,ServerEOS)` to be emitted.[2] Messages can also be sent to the handler, by tagging them with `i`.

In the Chat server, the handler will be very simple, namely the `transceiverF`:

```
transceiverF :: Socket -> F String (SocketMsg String)
```

A `transceiverF` will send any incoming messages to the socket, and emit strings from the socket as messages.

Now, we can have a look at our server program:

```
main = fudlogue (server (argKey "address" ""))

server addr = loopF (broadcast [] >^^=< socketServerF addr transceiverF)
  where
       broadcast cl = getSP $ \(i,e) ->
                      let clbuti = filter (/= i) cl
                          bc s cs = putSP [(i,show i++" "++s) | i <- cs]
                      in case e of
                        SocketNew -> bc "connected." clbuti $
                                     broadcast (i:cl)
                        SocketMsg s -> bc "says" cl $ broadcast cl
                        SocketEOS -> bc "has quit." clbuti $
                                     broadcast clbuti
```

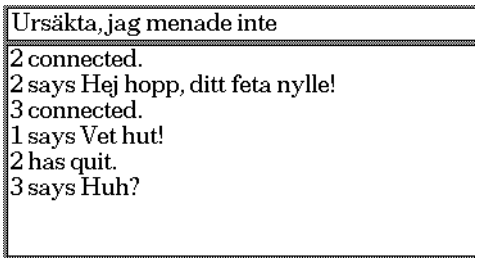

Figure 4: The Chat client.

2. `SocketMsg` should really be declared as a subtype of `ServerMsg`, if this was possible in Haskell.

The function `server` is illustrated in Figure 6. The stream processor `broadcast` maintains a list of active client numbers, and distributes messages from the `socketServerF`.

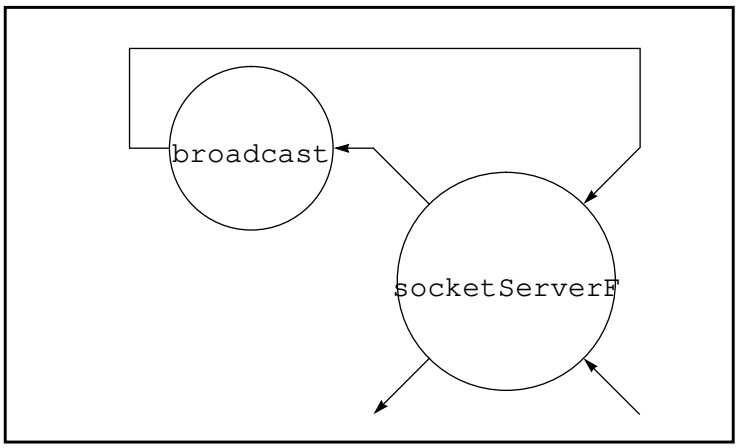

Figure 5: The `server` fudget

3.2.2 The client program

The visible part of the client program (seen in Figure 4) consists of an input field on top, realized by a `stringF`, and a fudget `terminalF`, that shows the incoming strings on subsequent lines. In the middle, we use `transceiverF` again, which handles the communication with the socket. The function `prep` of type `SocketMsg String -> String` extracts the strings from the socket messages. The first thing that `client` does is to connect to the server by means of

```
openSocketF :: SocketAddress -> (Socket -> F a b) -> F a b
```

which will emit the Haskell request `OpenSocket` and wait for the corresponding response `GotSocket`.

```
main = fudlogue (simpleShellF "Chat" [] None (client (argKey "address" "")))

client addr = openSocketF addr $ \s ->
              outF >==#< (5,LBelow,transceiverF s >==< inF)

inF :: F String String
inF = inputDoneF >==< stringF "" None

outF :: F (SocketMsg String) Char
outF = terminalF None defaultFont 20 50 >=^< prep
   where
         prep (SocketMsg s) = s
         prep SocketEOS = "The server died!"
```

The fudget structure of the client can be seen in Figure 6.

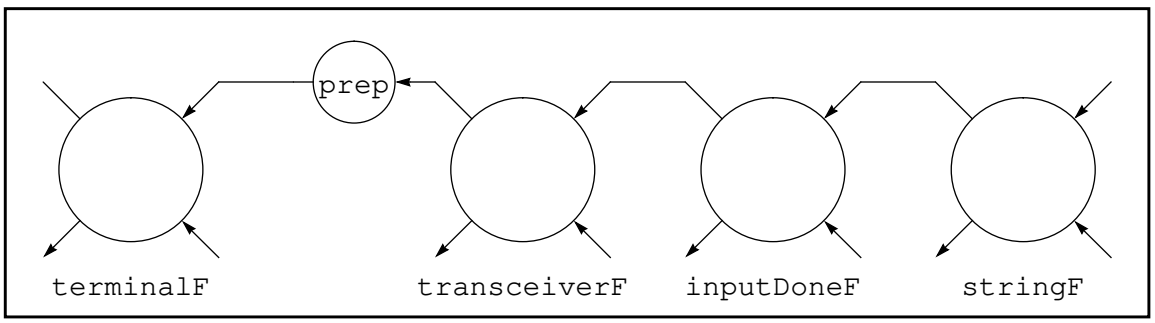

Figure 6: The `client` fudget

3.3 Implementation of `socketServerF`

To see an example how listener sockets are used, let us have a look at the implementation of `socketServerF`:

```
socketServerF :: LSocketAddress -> (Socket -> F a (SocketMsg b))
                                -> F (Int,a) (Int, ServerMsg b)
socketServerF addr handler = loopLeftF (idRightF (control >+< dynListF)
                                          >=^^< concmapSP router)
  where
      router e =
         let todyn = Inl . Inr
             out = Inr
         in
            case e of
             -- from control
             Inl (Inl (i,f)) -> [todyn (i,DynCreate f), out (i,ServerNew)]
             -- from dynListF
             Inl (Inr (i,m)) ->
               case m of
                SocketMsg m' -> [out (i,ServerMsg m')]
                SocketEOS    -> [out (i,ServerEOS), todyn (i,DynDestroy)]
             -- from outside
             Inr (i,m) -> [todyn (i,DynMsg m)]

      control = openLSocketF addr $ \lsocket ->
                putFu [Low (DoIO (AcceptSockets [lsocket]))] $
                (accepter 0)
         where accepter i = getFu $ \e -> case e of
               Low (AsyncInput (AISocketAccepted _ socket)) ->
                  putFu [High (i,handler socket)] $
                  accepter (i+1)
               _ -> accepter i
```

The `socketServerF` consists of a `dynListF` in parallel with the fudget `control`. There is also the stream processor `router` which will route messages from `dynListF`, `control`, and the outside to the right destination. See also Figure 7 for a diagram of `socketServerF`.

The `control` fudget starts by opening a listener socket by means of

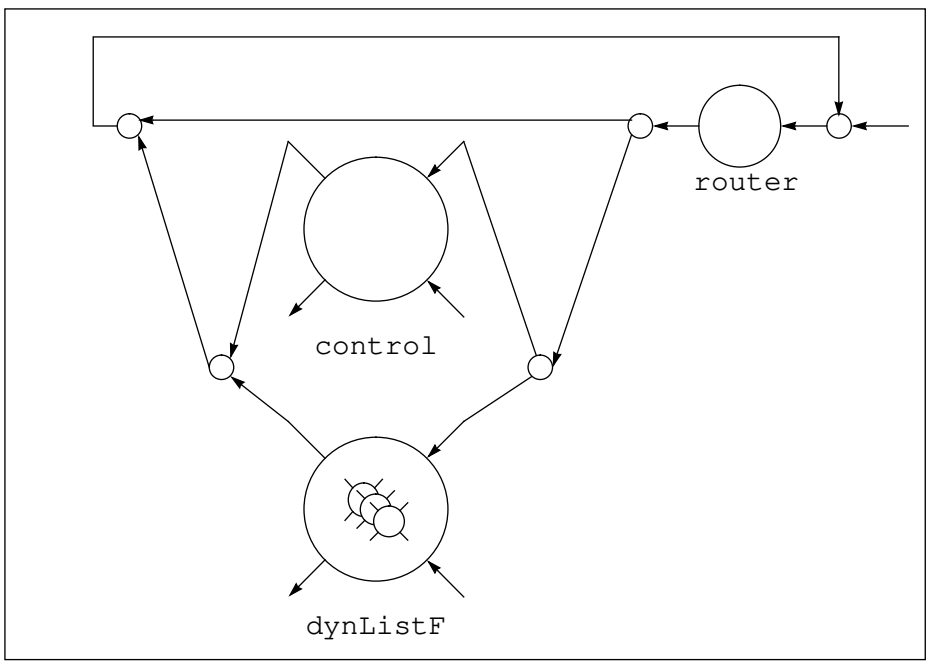

Figure 7: `socketServerF`. Inside `dynListF` are handlers for the accepted sock-

```
openLSocketF :: LSocketAddress -> (LSocket -> F a b) -> F a b
```

which emits the Haskell request `OpenLSocket` and waits for the response `GotLSocket`. Then, a `AcceptSockets` request is emitted. This request is not sent directly to the run time system, because there might be a number of fudgets in the program that want to accept sockets asynchronously. Therefore, `fudlogue` (which turns the top-level fudget into a `Dialogue` function) maintains a table which is the union of all listener sockets of the `AcceptSockets` requests that the different subfudgets have emitted, together with the paths of the subfudgets. `fudlogue` will emit an `AcceptSockets` request with all listener sockets from this table. When the asynchronous input value `AISocketAccepted ls s` is received, `ls` will be used to look up the path to the subfudget that wants the input.

But we do not have to bother about anything of this machinery when we look at `control`. It can perform its tasks without any knowledge about other fudgets in the program. After emitting the `AcceptSockets` request, it will enter the `accepter` loop, which waits for asynchronous input of form `AISocketAccepted`. When a new socket is accepted `control` applies `handler` to it and spits it out as a high level message, together with a fresh integer identifier. The `router` will direct this message to the `dynListF` fudget, which will start the new handler fudget.[3] `router` will also direct messages from the outside to the corresponding handler, and it will also destroy handlers when they emit `SocketEOS`.

3.4 Implementation of `transceiverF`

`transceiverF` consists of a transmitter and a receiver:

```
transmitterF :: Socket -> F String a
transmitterF s =
     loop (\l -> getFu $ \e -> case e of
           High str -> putFu [Low (DoIO (WriteSocket s (str++"\n")))] l
           _ -> l)

receiverF :: Socket -> F a (SocketMsg String)
receiverF s = stripSum >^=< idLeftF (SocketMsg >^=<
                                     absF (linesSP `serCompSP` concSP))
                         >==< read
  where
       read = putFu [Low (DoIO (ReadSockets [s]))] $
              loop (\l ->
                    getFu $ \e ->
                    case e of
                       Low (AsyncInput (AISocketRead _ str)) ->
                             if str == "" then putF [High (Inl SocketEOS)] l
                                          else putF [High (Inr str)] l
                       _ -> l)


transceiverF :: Socket -> F String (SocketMsg String)
transceiverF s = receiverF s >==< transmitterF s
```

There is no guaranty that a string sent to a socket will be received as one message in the other end. It might be received in smaller pieces. Therefore, the transmitter will append a line break as a separator to the string input as high level messages and write them to the socket. Similarly, the receiver will collect characters from the socket until a line break is seen. This is done by means of

```
linesSP :: SP Char String
```

3. `dynListF` understands the messages `(i,DynCreate f)`, which creates a new fudget, `(i,DynMsg m)`, which sends a message to an existing fudget, and `(i,DynDestroy)` which destroys the fudget created with tag `i`.

The receiver starts by sending out the `ReadSockets` request, which implies that the input to the socket turns up asynchronously. If the empty string is received, this means that the end of the stream is seen, and the receiver emits `SocketEOS`.

Finally, we combine `receiverF` and `transmitterF` by serial composition into `transceiverF`.

4 Future work

- To implement the socket interface and the asynchronous I/O system and integrate it with the Fudget system.
- To introduce parallel evaluation in the Fudget system.

5 Conclusions

As the Space Invaders example shows, the Fudget library is useful not only for programs with traditional, static user interfaces, but also for more dynamic interfaces with many animated objects. As with traditional fudget programs, the program structure with one fudget per animated object reflects very closely what you see on screen.

As the server of the Chat application shows, the Fudget library is useful not only for programs with graphical user interfaces, but for parallel applications in general.

References

[1] M. Carlsson & T. Hallgren, Fudgets - A Graphical User Interface in a Lazy Functional Language, in *FPCA 93' - Conference on Functional Programming Languages and Computer Architecture*, pages 321--330, June 1993.

[2] M. Carlsson & T. Hallgren, The Fudget distribution, Available by anonymous ftp from ftp.chalmers.se:/pub/haskell/chalmers/lml-0.999.?.lmlx.tar.Z.

[3] L. Edblom, D.P. Friedman, *Issues in Applicative Real-time Programming*, Technical Report no. 129, Computer Science Department, Indiana University, Bloomington Indiana, USA, 1982.

[4] T. Hallgren, *Introduction to Interactive Real-time Multi-user Games Programming in LML*, PMG memo 89, Dept. of Comp. Science, Chalmers, 1989

[5] J. Gettys & R.W. Scheifler, *Xlib - C Language X Interface*, MIT X Consortium Standard, Aug 1991.

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

[7] S.L. Peyton Jones and P. Wadler, *Imperative Functional Programming*, in Proceedings of the 1993 Conference on Principles of Programming Languages, 1993.