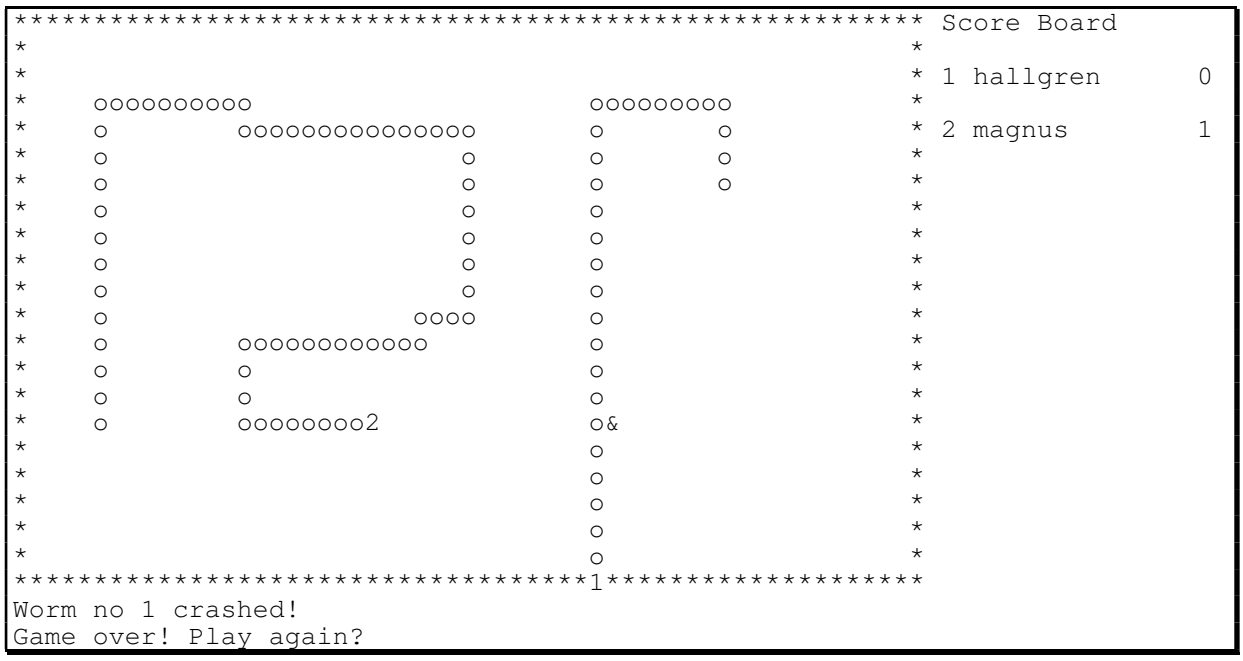
Introduction to Interactive Real-time Multi-user Games Programming in LML

Thomas Hallgren
Email: hallgren©cs.chalmers.se

ABSTRACT

No, this paper is not ment to be abstract.



Introduction

Until recently, if you wanted to implement an interactive real-time multi-user game, you had to do it in an imperative language. But now there is a lazy functional language which allows you to write such programs, namely LML [1].

In this paper I discuss how the features of LML make it possible to implement interactive real-time multi-user games. I present a simple such game, the Worm Game, which I implemented during the autumn term as some of the needed features became available in LML.

But first I should explain exactly what I mean with an interactive real-time multi-user game.

- A program is *interactive* if the program and the user interact, i.e. they can adjust their behaviour in accordance with the input they receive from each other. The user closes a loop of information flow from the output of the program to the input.
- A *real-time* program is a program that is affected by the flow of time in the real world, not just the input from the user. Not only *what* is input and output by the program is significant, but also *when* input and output occurs.
- A *multi-user* program is a program that interacts with several users at the same time and the users interact with each other through the program.

The purpose of this paper is not to present new theory on functional IO systems, but merely to illustrate how an existing system can be used.

The Worm Game

The picture on the front page illustrates what the screen typically looks like when you play the Worm Game.

In the Worm Game you control a worm moving on the screen. It is originally a single-user game, and the object is to see how long the worm can get. The longer the worm, the higher the score. The worm grows when it eats. There is always a piece of food somewhere on the screen, which the worm can eat by running over it. The worm moves forward at a constant speed and you control the worm by pressing keys to change directions. The game ends when the worm runs into a wall or itself.

In the multi-user variant, two (or more) players compete. You play several rounds. A round ends when a worm crashes. The worm(s) that didn't crash wins that round and gets a point. (There is lot of room for variation, of course.)

As may be apparent, the play field consists of a number of character squares that are either empty or occupied by a worm, the food or a wall. The worms move forward in discrete steps of one character square.

Interactive programs in LML

Like most other computer games, the Worm Game is interactive. So, how do you write interactive programs in LML?

There are several approaches to IO in functional programming, for example the stream based approach and the continuation based approach. Haskell [6] supports both of these approaches. The IO system of LML [3] is different from both of them.

In the simplest case, an LML program is a (pure) function from character lists to character lists. When a program is run, the run-time system applies it to the list of characters read from the terminal keyboard and prints the result on the terminal screen. So unlike programs in imperative languages, LML programs don't contain any explicit requests to perform input and output to the terminal.

Since LML is a lazy language, it is possible to write interactive programs. Part of the output can be produced when only part of the input has been read.

As an example, consider this simple program, which reads a list of numbers and outputs the sum of the numbers read so far, immediately after each number in the input has been read.

```
-- A simple interactive LML program
let readIntList = map stoi o choplist takeword

and printIntList = concmap (\n.itos n @ "\n")

and rec sum acc [] = []
     || sum acc (n.ns) = (acc+n).sum (acc+n) ns

in (printIntList o sum 0 o readIntList) input
-- input is the input character list and o is function
composition.
```

To write a more complex interactive program you need a convenient way to structure it. In the approach I have taken[1] programs are built by sequencing commands that, in addition to performing some computation, consume some input and/or produce some output. A command is a function whose argument is the list of characters not yet read by the program and whose result is the output produced together with the list of characters still unread by the command. The result may also contain a value to be passed to the following command in the sequence.

The sequencing operator can be viewed as a replacement for function composition, that allows the composed "functions" to perform IO operations as side effects, meaning that commands don't have to deal with the input and output character lists explicitly.

The resulting programs when using this approach look much like imperative programs (except that there aren't any global variables), maybe because this way of handling IO is similar to the way it is handled in the denotational semantics for an imperative language [5]. There is also a similarity to the parsing operators used to write parsers in a functional language [4]. After all, the input character stream to a program is a sentence of some language that the program must parse.

The sequencing operator and basic commands to produce output and consume input can be define like this in LML:

```
module  -- A simple module for IO support

infix "!";    -- ! is used as the infix sequencing operator
export !,getword,getInt,print,printInt,program;

rec p1 ! p2 = \in0.let (in1,out1)=p1 in0
           in let (in2,out2)=p2 in1
           in (in2,out1 @ out2)

and print s in0 = (in0,s)
and getword p in0 = let (item,in1)=takeword in0
            in p item in1

and getInt p = getword (p o stoi)
and printInt = print o itos

and program p =snd o p
end
```

The `get` functions remind a little of how IO is handled in the continuation model, in that they take a function which is applied to the result of the get operation.

The example program above can now be written like this:

[1] I'm probably not the first one to do it this way.

```
let rec sum acc= getInt (\n.printInt (acc+n) ! sum (acc+n))
in program (sum 0) input
```

There is a small problem, however. Consider this simple command sequence:

```
print "What's your name? " !
getword (\name. print ("Hello, " @ name))
```

You expect this program to first ask "`What's your name?`", then wait for the user to type something in, and then print "`Hello,`" followed by whatever the user typed in. However, since in a lazy language, evaluation is driven by the need to print, and the value of the variable `name` is not needed until after "`Hello,`" is printed, no input is requested from the terminal until then. It is possible to solve this problem by re-defining `getword` to force `name` to be evaluated before further output can be produced.

Real-time programs in LML

The worm game is a real-time program. So, how do you write real-time programs in LML?

When I think of functions, I think of input-output relations, something which has nothing whatsoever to do with time, so timing and functional programs seem like an impossible combination. But computation takes time, of course. You can use this fact and write functions with the single purpose of delaying output from the program. But this is a waste of processor time and can't provide accurate timing in a multi-tasking environment.

In LML, the run-time system provides lists whose elements takes a specified amount of time to evaluate. These lists are obtained through certain *system requests* [3]. Here is a simple program to illustrate how you can use this to control output to appear at well defined points in time. It displays a counter that counts how many seconds have elapsed since the program was started:

```
let printIntList = concmap (\n.itos n @ "\n")
and rec counter n (tick.ticks) = n.counter (seq tick (n+1)) ticks
and ticks = case syscall theSysToken (SRTickStream 1000)
            in SATimeStream ts: ts
            || _ : fail "can't get a tickstream"
            end
in printIntList (counter 0 ticks)
```

A special combinator `seq` is used in the definition of the function `counter`. It evaluates the first argument but discards its value and returns the second argument. It is used here to force the `tick` element to be evaluated although we are not interested in its value.

The first argument to the `syscall` function is a *system token*: you must "pay" with a system token for each request to the `syscall` function. System tokens are discussed in more detail in a later section. `syscall` handles many other types of system requests, some of which are described further in a later section.

Indeterministic choice in LML

Most real-time programs, including the Worm Game, need to wait for particular events to occur, for example a program may need to wait for data to arrive from an input device, or just wait until a particular point in time. Often, there is more than one source of events, and the order in which the events will occur is not known in advance, so the program must be prepared to respond to whatever event happens first. This is known as making an indeterministic choice. So, how do you make an indeterministic choice in LML?

Let's first see what mechanisms for indeterministic choice other (imperative) languages offer. First, how do you wait for a single event to occur? The

operations to check for events may be blocking or non-blocking. If the input operation is blocking all you have to do is invoke it once, and it will block wait until the event occurs. As an example, in Pascal you can write

```
read(c);            (* c is character variable *)
```

to wait for a character to appear from the input. If the input operation is non-blocking, you can use a busy-waiting loop to wait for an event. The dialect of BASIC used in a popular micro computer provides a `GET` command, which returns a string containing a character if one has arrived, otherwise it will return the empty string.

```
30 REM wait for a character
40 GET a$     :REM a$ is a string variable
50 IF a$="" THEN 40
60 REM here when a character has been read
```

With non-blocking operations it's easy to wait for one of several events to occur: you just include the checks for all the events in the busy-waiting loop. With blocking operations however, you can only wait for one event at a time. A special mechanism is needed to wait for one of several events. In Ada, a task can block wait for a call to an entry with the `accept` statement. To wait for one of several entries to receive a call you can use the `select` construction:

```
select
  accept A do … end;      -- A is an entry
or
  accept B do … end;      -- B is another entry
end;
```

Many other languages, like Pascal and C, has no built-in support for indeterministic choice, but there may be library routines or low-level operating system calls you can use instead.

Now then, how are these things done in LML? We have already seen examples of LML programs that wait for events. The first program, to sum numbers, waits for the user to type in numbers. As soon as an element in the input list is needed to compute more of the output, the program block waits until characters arrive. In the example with the real-time counter, the program block waits for the elements of the tick stream to become evaluated.

Since LML programs block wait for input events, a special mechanism is required for indeterministic choice. There is a special function provided for this purpose and it's called `choose`. Given two expressions, it computes both in parallel[1], and tells you which computation finished first. You are not restricted to choose between input events. For example, assuming we have two functions `fac` and `fib`, the result of

```
choose oracle (fac 10) (fib 7)
```

will be `true` if the computation of `fac 10` finished before the computation of `fib 7`, and `false` otherwise. The first argument to `choose` is an *oracle*, which `choose` needs to consult to do its job. The oracle will provide a useful answer the first time it is consulted. After that, it will always give the same answer as the first time. Oracles are discussed in more detail in a later section.

The following example is a complete LML program that combines the number summing example with the real-time example above. The counter counts up once every second and counts down every time the user hits a key on the terminal keyboard.

[1] On systems with only one processor, time-slicing will be used.

```
#include <SysCall>
#include <Oracle>
#include "oraclelists.t"  -- imports the oraclelists function

let rec merge (or.ors) xs ys=   -- general indeterministic merge
  let (us,vs)= if choose or xs ys
               then (xs,ys)
               else (ys,xs)
  in case us
     in [] : vs
     || u.us : u.merge ors us vs
     end

and ticks= case syscall theSysToken (SRTickStream 1000)
           in SATimeStream ts: reduce (\t.\ts.seq t (t.ts)) nil
ts
           || _ : fail "can't get a tick stream"
           end

and printIntList= concmap (\n.itos n @ "\n")

and rec sum acc (n.ns)= n+acc.sum (n+acc) ns

and ors._ = oraclelists oracletree   -- obtain a list of oracles

in CCBREAK.printIntList (sum 0 (merge ors (map (\c.-1) input)
                                          (map (\t.1) ticks)))
```

`theSysToken` is a system token supplied by the run-time system. There is a system request to get two tokens for one, so you can make more than one system call. `oracletree` is a tree of oracles supplied by the run-time system. The function `oraclelists` turns the oracle tree into a list of lists of oracles. The `merge` function is a very general and useful function.

Other useful features in LML

There is a number of other often useful features in LML.

- Termcap support: you can use cursor addressing, clear the screen, get the number of character rows and columns, and still keep the program terminal type independent.
- You can easily switch the terminal between cooked and raw mode.
- You can control the buffering of output to the terminal.
- You can access the command line arguments and environment variables.

The `syscall` function handles many useful system request. Amongst other things, you can:

- read the contents of a file,
- open a file for writing,
- read the system clock,
- get the identity of the user running the program,
- get the name of the machine the program is running on,
- get a directory listing,
- create and connect to sockets (for communication with other UNIX™ programs) (described further in a later section).

The Worm Game in LML

In the above sections I have introduced the features of LML needed (or at least useful) to implement a real-time game. They are all used in my implementation of the Worm Game.

There are two sources of events that control the output from the program:

- it is time to move the worm forward a step,
- the user presses a key to change the direction of the worm.

One convenient way to handle events from several sources is to merge them into one stream of events using the indeterministic `merge` function described in a previous section. I use a disjoint sum type to represent the elements of the merged event stream:

```
type InputEvent = Tick + Key Char
```

The code to generate a tick stream and merge it with the input from the user can be kept apart from the rest of the code. The main module of my implementation looks something like this (with some details omitted for readability):

```
let eventStream =
  let ticks = map (\t.Tick) tickStream
  and keys = map Key input
  in merge oracleList ticks keys
in CBREAK. program wormGame eventStream
```

Above, `tickStream` is a tick stream obtained with a system request as described in a previous section, and `wormGame` is the main function of the game proper, which initializes the game and calls the main loop. Somewhere inside the main loop the input events are analyzed with an ordinary `case` expression.

Apart from IO and timing, you also need data structures to represent the state of the game. For each worm you need to keep track of the character squares it occupies. Except when it eats, the length of the worm is constant. When the head of the worm moves a step forward, so does the tail. This suggest a FIFO queue as a suitable data structure to keep track of the worm. But you need more. Every time the worm moves forward a step you need to check if it has run into something, i.e. the head has reached a square already occupied by something else. This suggests that you need a table where you can look up the state of a character square one the screen. An updateable array seems to be the most suitable data structure for this. LML doesn't support arrays, however. You can of course use other data structures, such as binary search trees, but it turned out that simply testing if the new position of the head is a member of the queue representing the body of the worm is efficient enough to be useful.

The main loop of the game is a tail recursive function that maintains a data structure representing the current state of the game. Each revolution in the loop handles one input event.

The state of the game contains the state of the worm and a list of random numbers for the position of the food. The state of a worm is represented by the following type:

```
type WORM = Worm Int                    -- worm identity number
                 (Int#Int)              -- position of head
                 (QUEUE (Int#Int))      -- worm body
                 (Int#Int)              -- current direction
                 Int                    -- how much to grow
                 Int                    -- score
```

I won't describe the details of main loop further. I'm sure the reader can imagine the rest.

So far, I have only described the single-user version of the game. The extensions needed to allow several players are rather straight forward. The state of the game is changed to contain a list of worms instead of a single worm. When a tick event occurs, all the worms are moved forward, and for each worm you check if it has run into inself, another worm, the food or a wall. The `InputEvent` type is changed to

```
type InputEvent = Tick + Key Int Char
```

where the `Key` events are tagged with an integer identifying the user who originated the character.

But how do you get input from and present output to several players? The first very simple solution is to have all players (probably not more than two…) sit by the same terminal, use different sets of keys on the same keyboard, and watch the same screen. This means that the terminal will merge the input from different players into one stream for you. You only need an extra function to map the input characters into `Key` events which can then be merged with the tick stream to create the merged stream of input events as before.

A better solution is to let each player use his/her own terminal and have the game communicate with several terminals. The next section describes the LML features that allow you to do this.

Support for communication in LML

To be able to send output to more than one destination an LML program can, instead of just producing a character stream to be displayed on the terminal screen, produce a stream of *system commands*. One of the system commands is `SCPrChannel` which takes a *channel* and a character stream. The channel `c_stdout` (supplied by the run-time system) is used to print on the terminal screen. Some system requests return channels, for example `SRCOpenFile`, which opens a file for writing and returns the channel to which you print the new contents of the file. Here is a simple example program to store a line of text in a file:

```
#include <SysCall>
let myfile =
   case syscall theSysToken (SRCOpenFile "myfile")
   in SAChannel ch: ch
   || _ : fail "Error opening file for write"
   end
in [SCPrChannel myfile ("This text is stored in the file")]
```

One important property of the system command stream is that all the commands in it are executed in parallel. For example, a multi-user game usually needs send output to several terminals in parallel.

In UNIX™ there are facilities to allow programs to communicate with other programs running on the same or other machines. A server program can create a *socket* [7], and wait for client programs to connect to it. Once a connection between the server and a client has been established, data can flow in both directions.

There are system requests in LML to create sockets and to connect to sockets created by other programs. To connect to a socket, the `SRCOpenSocket` request is used:

```
syscall token (SRCopenSocket "birk::6000")
```

The string identifies the socket to connect to. The first part of the string is the name of the machine where the server program is running. If a connection was established, the result from this system request is a pair where the first component is the input stream of characters from the socket and the second component is a channel on which the output to the socket is to be printed.

Similarly, a socket can be created with the `SRCCreateSocket` request, which returns a stream of pairs of input streams and channels. Elements of this stream become available as connections are established.

Below follows a simple server program. It creates a socket and for each connection established it writes "`Client #`*n* `says: …`" on the terminal and send the message "`Hello! You are client #`*n*" to the client, where *n* is `1` for the first connection, `2` for the second, etc.

```
#include <SysCall>
let socket = "undis::6000"      -- must run on undis
in let connections =
   case syscall theSysToken (SRCCreateSocket socket)
   in SASocketStream connections: connections
   || _ : fail ("Error creating socket" @ socket)
   end
in let command ((fromclient,toclient),n) =
   let ns= itos n
   in [ SCPrChannel c_stdout ("Client #" @ ns @ "says:" @
                              fromclient @ "\n");
        SCPrChannel toclient ("Hello! You are client #" @ ns @
                              "\n")
      ]
in let commands = concmap command
                          (combine (connections,from 1))
in commands
```

And here is a simple client program that can connect to the above server:

```
#include <SysCall>
let socket = "undis::6000"      -- find server on undis
in let (fromserver,toserver) =
   case syscall theSysToken (SRCOpenSocket socket)
   in SASocket connection: connection
   || _ : fail ("Error opening socket " @ socket)
   end
in [    SCPrChannel c_stdout ("Server says:" @ fromserver);
        SCPrChannel toserver ("Hello there, server program\n")
   ]
```

My multi-user version of the Worm Game consists of a server program and a client program. Each player runs the client program on his/her own terminal. One player starts the server in addition to a client (this will be automated in a future version). The client program sends the input from the terminal keyboard to the server, and displays the input from the server on the terminal screen. The server contains the game proper and a main module that creates a socket, waits for the appropriate number of connections to be established, merges the input from the clients into an `EventStream` (as described above) and sends the output from the game to all clients.

Oracles and System Tokens

The function `choose` for indeterministic choice and the `syscall` function for system requests have one thing in common: they both have a mysterious extra argument. Why?

One very important property of pure functional languages is the referential transparency. This means that the result of a function depends on the *value* of the arguments, and nothing else.

But what about `choose`? It tells you which one of two expressions is faster to evaluate. Clearly, the result has nothing to do with the value of the arguments. It is possible for `choose` to produce different result even if the values of the arguments are the same. For example, the result of

```
choose oracle 40320 (fac 7)
```

will probably be `true`, while the result of

```
choose oracle (fac 8) 5040
```

will probably be `false`, although the arguments to `choose` are the same in both cases, since `fac`[1] `8` is equal to `40320` and `fac 7` is equal to `5040`.

It should be obvious by now that the oracle plays an important role. Without it, `choose` would break the referential transparency. But by supplying different oracles every time `choose` is used, it is OK for the result to be different even if the other two arguments are the same. Since the result of a function may only depend on the value of the arguments, it must be the value of the oracle that determines the result, somehow. It must be the oracle that has the answer to the question that can't otherwise be answered without breaking the referential transparency. And since the value of a particular oracle can't vary, you will obtain the same answer if use the same oracle more than once.

The problem with the `syscall` function is similar to the problem with `choose`. The result of a system request depends not only on the request itself, but also on the state of the real world. So, to preserve referential transparency an extra argument, the system token, that is normally different every time, allows the result to differ, even if the same request is made more than once. For instance, you can read the contents of a file once, and then on a later occasion read it again to see if it has changed.

Concluding remarks

Writing interactive real-time multi-user programs in a functional language is now feasible. LML has all the features needed to do this.

When you write an interactive program, the behaviour of the program is as important as the input-output relation. One of the strengths of functional programming languages is that they allow you to concentrate on the input-output relation and not worry about details such as the exact order in which computations are performed. But this strength is of no use when you want a program to behave a certain way.

Acknowledgements

I would like to thank Magnus Carlsson for proof-reading this paper, and Lennart Augustsson for providing the features in LML, whithout which it would be impossible to implement an interactive real-time multi-user game in LML.

References

[1] Lennart Augustsson & Thomas Johnsson, *Lazy ML user's manual*, Distributed with the LML compiler, Göteborg 1989

[2] Lennart Augustsson 1989, *Functional non-deterministic programming or How to make your own oracle*, PMG memo 66

[3] Lennart Augustsson 1989, *Functional Input using System Tokens*, PMG memo 72

[4] ??? *Laboration 2: semantik för ett imperativt programspråk*. (se [5])

[5] Kent Pettersson, *Semantik för programspråk*, föreläsningsanteckningar.

[6] Paul Hudak & Philip Wadler (eds) Dec 1988, *Report on the Functional Programming Language Haskell*

[7] ??? Sockets in UNIX™

[1] `fac` is the factorial function.