Chapter 1

Implementing Mobile Haskell

André Rauber Du Bois[1], Phil Trinder[1], Hans-Wolfgang Loidl[2]

Abstract: Mobile computation enables computations to move between a dynamic set of locations, and is becoming an increasingly important paradigm. *m*Haskell is a conservative extension of Haskell that supports mobile computation in open distributed systems. This paper outlines the *m*Haskell primitives, discusses the design and pragmatics of their implementation and includes preliminary performance comparisons with Jocaml. The implementation solves several challenges, including ensuring *predictable communication* in a lazy language with sharing, and using a combination of bytecode and machine code to manage the *common software base*, i.e. to determine what to communicate between locations.

1.1 INTRODUCTION

Mobile Haskell [BTL03] (*m*Haskell) is a conservative extension of the purely functional Haskell language designed to facilitate the construction of distributed mobile software. As depicted in Figure 1.1, *m*Haskell extends Concurrent Haskell [JGF96], an extension supporting concurrent programming, with higher order communication channels called *Mobile Channels* (MChannels), that allow the communication of arbitrary Haskell values including functions and channels.

The main features of the *m*Haskell implementation are:

- *m*Haskell *supports the construction of open systems*, enabling programs to connect and communicate with other programs and to discover new resources in the network. The abstractions we use to provide this basic functionality are MChannels and remote evaluation. They have fast implementations in the RTS (runtime system) using C and TCP/IP sockets.

[1]School of Mathematical and Computer Science, Heriot-Watt University, Edinburgh EH14 4AS, Scotland, Email:{`dubois,trinder`}`@macs.hw.ac.uk`

[2]Ludwig-Maximilians-Universität München, Institut für Informatik, D 80538 München, Germany, Email: `hwloidl@informatik.uni-muenchen.de`

- *m*Haskell *is portable*. It is implemented as an extension of the GHC (*Glasgow Haskell Compiler*) [GHC98] compiler that has been ported to many different architectures and operating systems. Our extensions are implemented using standard C and TCP/IP sockets, maintaining a high degree of portability.

- *m*Haskell *is designed to run on heterogeneous networks*. Mobile languages designed to work on global distributed systems, such as the Internet, must be able communicate code between machines of different architectures and operating systems. The usual approach for communicating computations on heterogeneous networks is by compiling programs into architecture independent byte-code. GHC is both an optimising compiler and an interactive environment called GHCi, which compiles user defined functions into byte-code, and this technology could be used by *m*Haskell for communicating computations on heterogeneous networks.

- *m*Haskell takes a *hybrid approach*, combining byte-code and machine code. GHCi is designed for fast compilation and linking, it generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. As the basic modules in GHC are compiled into machine code and are present in every standard installation of the compiler, the routines for communication have to send only the machine independent part of the program and link it to the local definitions of the machine dependent part when the code is received. This gives us the advantage of having much faster code than using only byte-code.

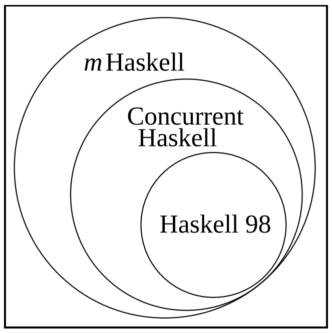

FIGURE 1.1. *m*Haskell is an extension of Concurrent Haskell

This paper is organised as follows: In the next section we present the MChannels communication primitives and the primitives for resource discovery and registration. In section 1.3 the implementation of *m*Haskell is described first by giving a general overview of the platform and its challenges and then by describing each of the design decisions.

1.2 MOBILE HASKELL

1.2.1 Communication Primitives

Figure 1.2 shows the MChannel primitives. Haskell with Ports [FH01] has similar primitives but restricts the type of values that can be communicated to basic values and data types, no functions or IO computations can be communicated.

```
data MChannel a      -- abstract
type HostName = String
type ChanName = String

newMChannel      :: IO (MChannel a)
writeMChannel    :: MChannel a -> a -> IO ()
readMChannel     :: MChannel a -> IO a
registerMChannel :: MChannel a -> ChanName -> IO ()
unregisterMChannel:: MChannel a -> IO()
lookupMChannel   :: HostName -> ChanName ->
                           IO (Maybe (MChannel a))
```

FIGURE 1.2. Mobile Channels

The `newMChannel` function is used to create a mobile channel and the functions `writeMChannel` and `readMChannel` are used to write/read data from/to a channel. MChannels are synchrous and have similar semantics to Concurrent Haskell channels: when a value is written to a channel the current thread blocks until the value is received in the remote host. In the same way when a `readMChannel` is performed in an empty MChannel it will block until a value is received on that MChannel. The functions `registerMChannel` and `unregisterMChannel` register/unregister channels in a name server. Once registered, a channel can be found by other programs using `lookupMChannel` which retrieves a mobile channel from the name server. A name server is always running on every machine of the system and a channel is always registered in the local name server with the `registerMChannel` function. MChannels are single reader channels and values are evaluated to normal form before being communicated.

In figure 1.3 a simple example describing how to use MChannels is presented. Fist a program running on a machine called `ushas` registers a channel `mv` with the name `"myC"` in its local name server. When registered the channel can be seen by other machines using the `lookupMChannel` primitive. After the lookup, the connection between the two machines is established and communication is performed with the functions `writeMChannel` and `readMChannel`.

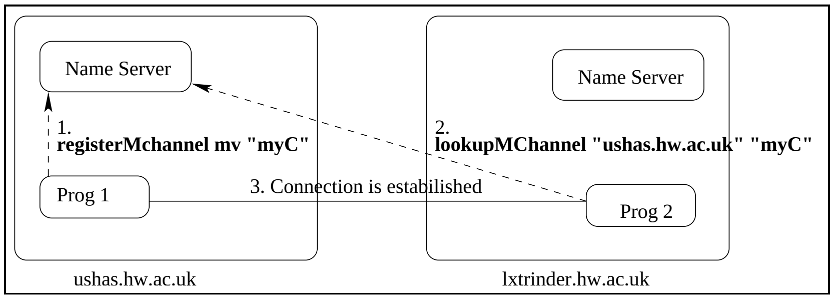
Name Server
Name Server
1.
registerMchannel mv "myC"
2.
lookupMChannel "ushas.hw.ac.uk" "myC"
Prog 1
3. Connection is estabilished
Prog 2
ushas.hw.ac.uk
lxtrinder.hw.ac.uk

FIGURE 1.3. Example using MChannels

1.2.2 Discovering Resources

One of the objectives of mobile programming is to better exploit the resources available in a network. Hence, if a program migrates from one node of the network to another, this program must be able to discover the resources available at the destination. By resource, we mean *anything* that the mobile computation would like to access in a remote host, from simple files to databases.

```
type ResName = String

registerRes :: a -> ResName -> IO ()
unregisterRes :: ResName -> IO ()
lookupRes :: ResName -> IO (Maybe a)
```

FIGURE 1.4. Primitives for resource discovery

Figure 1.4 presents the three *m*Haskell primitives for resource discovery and registration. All machines running *m*Haskell programs must also run a registration service for resources. The scope for the primitives for resource discovery is only the machine where they are called, thus they will only return values registered in the local server. The `registerRes` function takes a name (`ResName`) and a resource (of type `a`) and registers this resource with the name given. `unregisterRes` unregisters a resource associated with a name and `lookupRes` takes a `ResName` and returns a resource registered with that name. To avoid a type clash, if the programmer wants to register resources with different types, she has to define an abstract data type that will hold the different values that can be registered.

A better way to treat these possible type clashes would be to use dynamic types like Clean's *Dynamics* [Pil98], but at the moment there is no complete implementation of it in any of the Haskell compilers.

1.2.3 Remote Thread Creation

*m*Haskell also provides a construct for remote thread creation:

```
rforkIO :: IO () -> HostName -> IO ()
```

It is similar to Concurrent Haskell's `forkIO` as it takes an IO action as an argument but instead of creating a local thread it sends the computation to be evaluated on the remote host `HostName`. The `rforkIO` function is implemented using MChannels as described in [BTL03].

1.2.4 A Simple Example

Figure 1.5 shows a *m*Haskell program that visits a `listomachines` and executes the computation called `mobile` on all the machines of the list. First a channel `mch` is created and registered with the name `"mainmch"`, this channel is used by the remote locations to send the result of the computation back to the main machine. Then, the function `sendMobile` is mapped over the `listofmachines`, this function, looks for a specific channel called `clientmch` on the remote host, and sends `mobile` to be executed remotely. The client receives the computation, executes it and sends the result back to the main program through the `mch` channel.

This program, although simple, uses all the facilities provided by *m*Haskell (i.e. remote MChannels, Registration of resources and mobile computation), and is used in the measurements given in section 1.5

1.3 IMPLEMENTATION DESIGN

1.3.1 Introduction

Mobile systems must abstract over the heterogeneity of large scale distributed systems, allowing machines of different architectures running different operating systems to communicate. This abstraction is usually achieved by compiling programs into architecture independent byte-code. As a platform to build our system, we have chosen the *Glasgow Haskell Compiler* (GHC) [GHC98], a state of art implementation of Haskell. The main reason for choosing GHC is that it supports the execution of byte-code combined with machine code. GHC is both an optimising compiler and an interactive environment called GHCi. GHCi is designed for fast compilation and linking, it generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. Both GHC and GHCi share the same runtime-system, based on the Spineless Tagless G-machine (STG)-machine [Jon92], that is a graph reduction machine.

This and the next section explain how the implementation of *m*Haskell using the GHC compiler works. First, in this section, we discuss some design options in the language level, such as the evaluation of thunks (unevaluated expressions), and how to deal with shared computations and MChannels, and how these design options influence a real implementation. In the next section we discuss the low

```
main = do
  mch <- newMChannel
  registerMChannel mch "mainmch"
  list <- mapM (sendMobile mobile mch) listofmachines
  let v = sum list
  print ("Total Load of the network: " ++ (show v))
  where
   mobile = do
     res <- lookupRes "getLoad"
     case res of
     Just getLoad -> do
                     load <- getLoad
                     return load
     Nothing      -> return 0
   listofmachines = (...)

sendMobile :: IO () -> MChannel Int -> HostName -> IO Int
sendMobile comp mch host = do
  mc <- lookupMChannel host "clientmch"
  case mc of
    Just nmc -> writeMChannel nmc comp
  result <- readMChannel mch
  return result
```

FIGURE 1.5. Program that computes the load of a network

level issues of the implementation such as how to force the evaluation of expressions, packing routines and the implementation of MChannels.

1.3.2 Evaluating Expressions before Communication

When a value is sent through a channel, it is evaluated to *normal form* before communication occurs. The reason for this design decision is that *m*Haskell should have predictable communication behaviour, since in a mobile application the amount of communication and placement of work and data is crucial. To clarify the possible complications for predicting such placement, consider the following example:

```
let
 (a,b,c) = f x
in
if a then
  writeMChannel ch b
```

Suppose that in the tuple returned by `f x` the first value `a` is a Boolean, the second `b` an integer, and the third `c` is a really large data structure (i.e. a big tree). Based on the value of `a` we choose if we want to send the integer `b` (and only `b`) to a remote host. In the example, it seems that the only value being sent is the integer, but because of lazy evaluation that is not what happens. In the beginning of the evaluation, we have a graph similar to the one in figure 1.6.

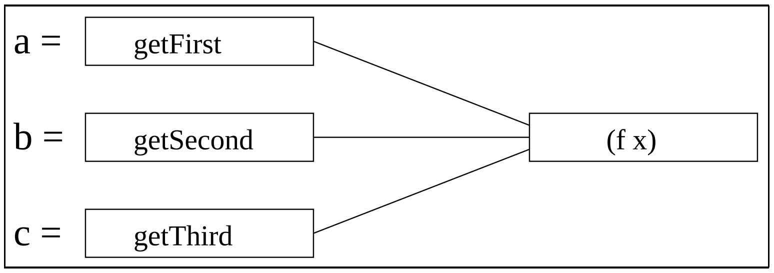

FIGURE 1.6. Graph for `let (a,b,c) = f x`

At the point where `writeMChannel` is performed, the value `b` is represented in the heap as the *selector* that gets the second value of a tuple applied to the whole tuple. If `writeMChannel` does not evaluate its argument before communication, the whole value is communicated and is difficult to see that in the Haskell code.

The evaluation of thunks still allows the programmer to send IO computations and functions that wont be affected by an evaluation using `seq` (Haskell function that evaluates its argument to *weak head normal form*), and this evaluation to normal form will not affect programs using *remote thread creation* because all computations started using `rforkIO` are in fact IO actions.

There are still ways of sending pure expressions to be evaluated on remote hosts. The programmer can send a tuple with a function and its arguments, and the function is applied to the values only on the remote end. Unevaluated expressions can also be communicated if wrapped in an IO value, as in the `apply` function:

```
apply (a->b) -> a -> IO b
apply f x = return (f x)
```

1.3.3 Sharing Properties

Many non-strict functional languages are implemented using graph reduction, where a program is represented as a graph and the evaluation of the program is performed by rewriting the graph. The graph ensures that shared expressions are evaluated only once [PJ92].

Maintaining sharing between nodes in our distributed system will result in a large number of extra-messages and call-backs to the machines involved in the computation (to request structures that were being evaluated somewhere else or to update these structures) that the programmer of the system did not know about. Again, predictability of communication may be lost. In a typical mobile application, the client will receive some code from a channel and then the machine can be disconnected from the network while the computation is being executed (consider a handheld or a laptop). If we preserve sharing, it is difficult to tell when a machine can be disconnected, because even though the computation is not being executed anymore, the result might be needed by some other application that shared the same graph structure. The problem is partially solved by making the primitives strict: expressions will be evaluated just once and only the result is communicated.

1.3.4 MChannels

MChannels are single reader channels. There are two main reasons for making them single reader. First, it is difficult to decide where a message should be sent when we have more than one machine reading values from the same channel. The main question is where is this channel located? To implement channels with multiple readers we would need to maintain some sort of distributed state keeping track of all the machines that have references to the channel and these references must be updated every time the channel is moved to another place.

A simple way to implement multiple reader channels would be to keep the channel in one place, the place where it was created, and all other references to the channel read and write values into the channel by sending messages to this main location. The problem with this approach is that if the main location crashes all the other machines that have references to the channel cannot communicate anymore (Figure 1.7).

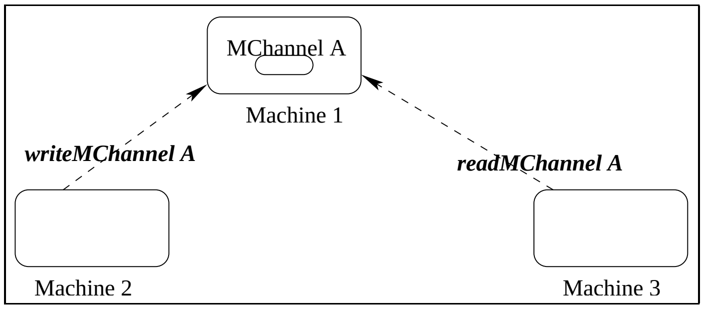

FIGURE 1.7. Machines 2 and 3 cannot communicate if Machine 1 crashes

The second reason for having single reader channels is security: with multiple reader channels one process can *pretend* to be a server and steal messages. This is a classic problem also found in the untyped π-calculus [Mil99].

1.4 THE IMPLEMENTATION

1.4.1 Packing Routines

The graph representing the computation being communicated is packed at the source and unpacked at the destination. The *m*Haskell pack and unpack routines are based on the GUM [THM$^+$96] system, but are extended to pack GHCi's Byte-Code Objects (BCOs).

Packing, or *serialising*, arbitrary graph structures is not a trivial task and care must be taken to preserve sharing and cycles. As in GPH [THM$^+$96], GDH [PTL00] and Eden [BLOMP97], packing is done breadth-first, closure by closure and when the closure is packed its address is recorded in a temporary table that is checked for each new closure to be packed to preserve sharing and cycles. We proceed packing until every reachable graph has been packed.

The main heap object to be packed in our implementation of *m*Haskell is the BCO, that is GHC's internal representation for its architecture independent byte-code. A BCO is composed by its `info_table` (that contains information about the closure's fields and also its entry code), a list of instructions, a list of pointers and a list of info tables. The BCO's info table is the same for every BCO so it does not need to be packed, its list of instructions is just a list of bytes and is packed easily. The list of pointers contains a list of other closures that are used in the byte-code instructions hence all of them must also be packed. The list of info tables contains pointers to info tables of data structures that are constructed during the execution of the BCO's instructions. Those info tables are machine dependent hence are packed in a special way explained in section 1.4.2.

As the basic modules that come with GHC are compiled into machine code and are present in every standard installation of the compiler, the packing routines have to pack only the machine independent part of the program and link it to the local definition of the machine dependent part when the code is received and unpacked. This gives us the advantage of having much faster code than using only byte-code. Once packed, the BCO can be communicated in the way described in section 1.4.4. All machines running the mobile programs should have the same version of the GHC/GHCi system with an implementation of the primitives for mobility and also have the same binary libraries installed. Programs that communicate functions that are not in the standard libraries must be compiled into byte-code using GHCi.

Our packing mechanism gives us a simple way of controlling the amount of code communicated: since only functions that are compiled into byte code are packed, if the programmer knows that one module used in the computation is already in the remote host, this module must be compiled into machine code, thus it will not be communicated.

Programs that will only receive byte-code do not need to have a GHCi installed because the byte-code interpreter is part of GHC's RTS. In fact, if only functions from the standard libraries are used in the mobile programs, there is no need to have GHCi at all in both ends of the communication.

1.4.2 Communicating user defined types (ADTs)

Currently user defined types are always compiled into machine code in GHCi. There are two ways to overcome this problem. The first one would be to compile the types into a different type of closure that uses BCOs internally. This requires changing the compiler. The other solution is to ship the data type including the values in its info table. The entry code for these objects is very simple and has to be generated again in the destination.

In our current implementation, all data types used in the mobile programs must be defined in all the machines that are going to receive the code. Thus we only pack the name representing its info table in the linker and the content of its fields. When unpacking, we look for the local definition of the info table by searching for its name in the linker's tables. We consider an implementation of one of the two solutions described above, as a tuning step in the development of the prototype implementation, aiming to reduce the common software base needed on all machines.

1.4.3 Evaluating Expressions

A simple way to evaluate thunks would be to use *evaluation strategies* [THLP98], e.g.:

```
let list = [1..100]
in writeMChannel mch list
```

where in the definition of `writeMChannel` we use the `rnf` strategy to evaluate its argument to normal form.

But strategies will not work in all cases. Consider the following example:

```
f:: a -> b -> Int

let
  a = (...)
in
writeMchannel ch (f a)
```

In this case is not possible, inside of the definition of `writeMChannel`, to evaluate the expression `a` using strategies. One solution to this problem would be to implement a function `kids` with type:

```
kids:: HValue -> Array# HValue
```

That takes a value from the heap (the expression to be evaluated) and return an array with all the thunks pointed by this value. Using `kids` we can write a `deepSeq :: a -> ()` function that recursively applies `seq` to all the thunks pointed by its argument.

Another way to evaluate thunks is to do it inside the RTS: using a primitive function that creates a new RTS thread to evaluate its argument to normal form by forcing the evaluation of all the expressions pointed by the argument.

In our implementation we use an hybrid approach: a thunk in the top level of the graph representing the computation is forced by a `seq` (as in figure 1.8). If there are other thunks in the graph, these thunks are evaluated by an extra thread in the RTS. Care must be taken to preserve the queue of closures yet to be packed if the new thread induces garbage collection. In short the packing queue is made visible to the Garbage Collector.

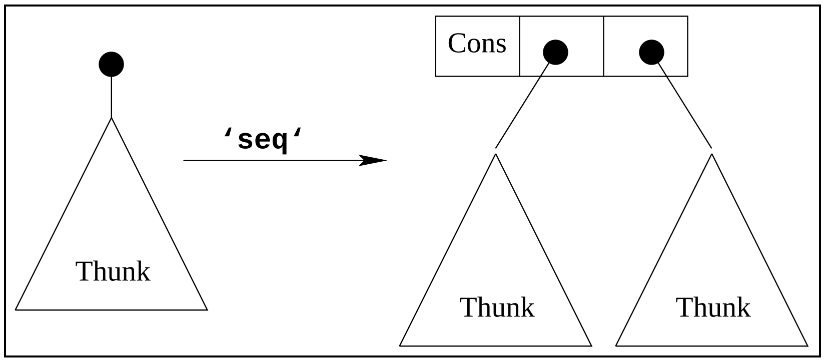

FIGURE 1.8. Evaluation of thunks using `seq`

1.4.4 Implementation of MChannels

The basic structure to support MChannels is implemented in a similar way to Ports in Distributed Haskell [VF01].

Communication is implemented using the standard sockets library provided by the operating system, thus avoiding the need of any extra libraries e.g. PVM or MPI. Haskell objects are serialised using the packing routines explained before and converted into an array of bytes that can be easily communicated through a socket.

Communication via sockets can be done using two different protocols: TCP and UDP. UDP is a fast connection less protocol that does not handle message loss. TCP on the other hand is a connection based protocol, making it easier to implement communication with the cost of a little extra overhead. We have chosen to implement the communication routines using TCP.

The channel data type is a simple Haskell data type that contains internally all the information that will be needed for communication, i.e. the name of the channel, the name of the host where it belongs and a concurrent Haskell channel (CHC) through which the communication between the program and the mobile runtime system occurs. When a new MChannel is created also a CHC is created to serve as a communication link between the program and the communication layer of the RTS. When a value is written into a MChannel, it is in fact written into its CHC. The RTS then reads this value from the CHC, serialises it and communicates it to

TABLE 1.1. Comparative Jocaml and *m*Haskell Execution Times

Number of Machines visited	Jocaml (sec)	*m*Haskell (sec)
1	0.05s	0.47s
2	0.06s	0.93s
4	0.10s	1.85s
8	0.16s	3.70s
16	0.28s	7.42s

the appropriate host based on the information present in the MChannel data type. When the RTS receives a value from a remote host this value is written into the CHC that represents the MChannel that should receive the message. A thread that reads a value from a MChannel is in fact reading a value from the internal CHC and will stay blocked in this CHC until a value is written by the RTS there.

To make ports visible to other machines in the network we use the `register-MChannel` and `lookupMChannel` primitives. These primitives communicate with an external naming service that keeps listening for requests on a well known port. This service maintains a table with all the ports registered in the machine in which it is running. It also communicates with lookups launched by other hosts looking for channels. When a lookup is received, all the information about the channel is sent back to the client so the client can communicate directly with the program that is waiting for requests on that channel.

1.5 INITIAL EVALUATION

Table 1.1 shows a comparison between Jocaml [CF99] and *m*Haskell using the mobile program from section 1.2.4.

Jocaml [CF99] is an extension to Objective-Caml [OCa02], a strict functional language with extensions for object oriented programming, used to develop systems with mobile agents. Jocaml extends Objective-Caml with a small set of primitives taken from the Join-Calculus [FG96]. Jocaml programs communicate and synchronise through messages sent on channels, called *names* in the Join-Calculus terminology.

Although *m*Haskell presents a good scalability when the number of machines is increased, it is still roughly twenty times slower than Jocaml when running on a system with more then two machines. The main reason for that is the routine that recursively transverses the graph forcing the evaluation of thunks before packing. Every time a computation is sent, the graph has to be transversed twice: once to force the evaluation and once for packing. It is not an option to force the evaluation while packing because the evaluation of the graph might change what has been already packed. Because Jocaml is strict, the evaluation of expressions to be communicated occurs naturally. Moreover, Jocaml is build as an extension to the Objective Caml compiler [OCa02], a compiler with primitives for serialisation.

*m*Haskell is still in its early stages and a lot of optimisation could be applied. For example, in the program used in the experiments, the same function is sent to different hosts and is repacked every time it is communicated. Such packed computations could be stored for reuse.

1.6 RELATED WORK

There are numerous parallel and distributed Haskell extensions [TLP02].

GPH and Eden are simple and powerful extensions to the Haskell language for parallel computing. They both allow remote execution of computation, but the placement of threads is implicit. The programmer uses the `par` combinator in GPH, or process abstractions in Eden, but where and when the data will be shipped is decided by the implementation of the language.

GDH is closer to the language presented here. Communication can be implemented using MVars and remote execution of computations is provided with the `revalIO` (remote evaluation) and `rforkIO` primitives. The problem in using GDH for mobile computation is that it is implemented to run on closed systems. After a GDH program starts running, no other PE (*processing element*) can join the computation. Moreover the GDH implementation relies on a *virtual shared heap* that is shared by all the machines running the computation. The algorithms used to implement this kind of structure will not scale well for very large distributed systems like the Internet [BTL03].

Haskell with ports is a very interesting model to implement distributed programs in Haskell because it was designed to work on open systems. The only drawback is that the current implementation of the language restricts the values that can be sent through a port to the basic types and types that can instantiate the `Show` class. Furthermore, the types of the messages which can be received with `readPort` must be an instance of the `Read` class. The reason for these restrictions is that the values of the messages are converted to strings in order to be sent over the network [FH01].

There are other extensions to functional languages that allow the communication of higher-order values. Kali-Scheme [CJK95] and Erlang [Erl02] are examples of strict untyped languages (Erlang is dynamically typed) that allow the communication of functions. Haskell is a statically typed language hence the communication between nodes can be described as a data type and many mistakes can be caught during the compilation of programs. Other strict typed languages such as Nomadic Pict [Woj00], Facile [Kna95] and Jocaml [CF99] implement the communication primitives as side effects while we integrate them to the IO monad hence preserving referential transparency.

Curry [Han99] is a functional logic language that provides communication based on Ports in a similar way to the extension presented in this paper. Goffin [CGK98] is a Haskell extension for concurrent constraint programming using ports but there is no distributed implementation of the language available yet. Another language that is closely related to our system is Famke [vWP02]. Famke is an implementation of threads for the lazy functional language Clean [NSvEP91] (using monads and continuations), together with an extension for distributed communication using ports. Famke has only a restricted form of concurrency, providing interleaved execution of atomic actions using a continuations monad.

1.7 CONCLUSIONS AND FUTURE WORK

We have presented the implementation of *m*Haskell, an extension of Haskell for building open, distributed mobile systems. Unlike related systems *m*Haskell can communicate arbitrary values, including functions and MChannels, between processors. This enables the use of powerful abstraction mechanisms provided by functional languages. Although the current implementation of *m*Haskell is still a prototype, it demonstrates the use of such abstraction mechanisms.

There are a number of issues that could be investigated in the future:

- It may be possible to extend the compiler with a *mobility analyses* (maybe based on a non-determinism analyses [PS02]) that would decide the parts of the program that should be compiled into byte-code and the parts that could be compiled into machine code, based on the occurrences of `writeMChannel`, as in [Kir01].
- The implementation could be optimised, e.g. maintain a cache of functions already communicated to avoid repeated communication.
- Some languages that support mobility of code also support the migration of running computations (usually referred as *strong mobility*). We could also extend Haskell with a primitive for transparent strong mobility, that would be a primitive to explicitly migrate threads:

  ```
  moveTo :: HostName -> IO()
  ```

  The primitive `moveTo` receives as its argument a `HostName` to where the current thread should be moved.

  Strong mobility could be implemented in two ways: RTS level and Code Transformation.

  – RTS level: We use packing routines that pack the state of the current thread (its stack) and sends it to be evaluated in a remote host. This work would extend our previous work on thread migration for the parallel functional language GPH [BLT02].

  – Code Transformation: During compilation a program using `moveTo` is transformed into a simpler program that uses only weak mobility. One way to do that is to lift the IO monad into a continuation monad and then every call to `moveTo` is translated into a remote evaluation of the continuation of the current thread.

ACKNOWLEDGEMENTS

The authors would like to thank Simon Marlow and Simon Peyton Jones for their helpful comments on the implementation design for *m*Haskell. Bernard Pope also gave important suggestions about the evaluation of thunks. This work has been partially supported by an ORS and James Watt Scholarship.

REFERENCES

[BLOMP97] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. The Eden Coordination Model for Distributed Memory Systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*, volume 1123. IEEE Press, 1997.

[BLT02] André Rauber Du Bois, Hans-Wolfgang Loidl, and Phil Trinder. Thread migration in a parallel graph reducer. In *IFL*. Springer-Verlag, LNCS 2670, 2002.

[BTL03] Andre R. Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Towards a Mobile Haskell. In *Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pages 113–116, Valencia (Spain), 2003.

[CF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[CGK98] Manuel M. T. Chakravarty, Yike Guo, and Martin Kohler. Distributed haskell: Goffin on the internet. In *Fuji International Symposium on Functional and Logic Programming*, pages 80–97, 1998.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5):704–739, 1995.

[Erl02] Erlang. WWW page, June 2002.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Conference on Lisp and Functional Programming (LFP'84)*, Austin, Texas, 1996.

[FH01] Volker Stolz Frank Huch. Distributed programming in haskell: From ports to streams. In *Haskell Workshop*, 2001.

[GHC98] The Glasgow Haskell Compiler. WWW page, January 1998.

[Han99] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702, pages 376–395. Springer LNCS, 1999.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[Kir01] Zeliha Dilsun Kirli. *Mobile Computation with Functions*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2001.

[Kna95] Frederick Colville Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie mellon University, 1995.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The* π*-Calculus*. Cambridge University Press, May 1999.

[NSvEP91] Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent Clean. In Leeuwen Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, volume 505, pages 202–219. Springer-Verlag, 1991.

[OCa02] OCaml. WWW page, June 2002.

[Pil98] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185, 1998.

[PJ92] S.L. Peyton Jones. *Implementation of Functional Programming Languages. A Tutorial*. Prentice Hall, 1992.

[PS02] R. Peña and C. Segura. A polynomial cost non-determinism analysis. In *Implementation of Functional Languages*, LNCS, Volume 2312, pages 121–137. Springer-Verlag, 2002.

[PTL00] R. Pointon, P.W. Trinder, and H-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *IFL*. Springer LNCS, 2000.

[THLP98] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[THM$^+$96] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.

[TLP02] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and distributed haskells. *Journal of Functional Programming*, 12(4/5):469–510, 2002.

[VF01] V.Stolz and F.Huch. Implementation of Port-based Distributed Haskell. In *Draft. Proc. of IFL*, 2001.

[vWP02] Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In *IFL 2002*, 2002.

[Woj00] Pawel Tomasz Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Wolfson College, University of Cambridge, 2000.