Chapter 1

Implementation of Mobile Haskell (DRAFT)

André Rauber Du Bois¹, Phil Trinder¹, Hans-Wolfgang Loidl²

Abstract: We present in this paper the implementation of mHaskell (Mobile Haskell), an extension to the pure functional language Haskell for building distributed mobile systems. mHaskell, unlike other extensions of Haskell, supports the communication of any value of the language (including functions and IO actions) in an open distributed system, i.e. a system where multiple executing programs can interact using a predefined protocol. The main features of our implementation are the ability of communicating on heterogeneous networks and the use of byte-code together with machine code for faster execution of mobile computations.

1.1 INTRODUCTION

This paper presents the implementation of Mobile Haskell [BTL03], or simply mHaskell, an extension to the pure functional language Haskell for the implementation of distributed mobile software. mHaskell extends Concurrent Haskell [JGF96] (see figure 1.1), another Haskell extension for concurrent programming, with the idea of higher order communication channels called Mobile Channels or MChannels. These channels allow the communication of any Haskell value including functions, IO actions and channels.

The main features of our implementation are:

- mHaskell is a language designed to work on open system. It provides ways for programs to connect and communicate with other programs and to discover

¹School of Mathematical and Computer Science, Heriot-Watt University, Edinburgh EH14 4AS, Scotland. Email:{dubois,trinder}@macs.hw.ac.uk
²Ludwig-Maximilians-Universität München, Institut für Informatik, D 80538 München, Germany. Email: hwloidl@informatik.uni-muenchen.de
new resources in the network using simple abstractions such as MChannels and remote evaluation. These abstractions have a fast implementation in the RTS (runtime system) using C and TCP/IP sockets.

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

- **mHaskell 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 a interactive environment called GHCi, which compiles user defined functions into byte-code, and this technology is used by mHaskell for communicating computations on heterogeneous networks..

- **Hybrid approach: byte-Code + 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. mHaskell is an extension of Concurrent Haskell](image)

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 mHaskell is described first by giv-
ing a general overview of the platform and its challenges and then by describing each of the design decisions in separate.

1.2 MOBILE HASKELL

1.2.1 Communication Primitives

In figure 1.3 is possible to see the MChannel primitives. These primitives are similar to those of Haskell with Ports [FH01], but without restricting the types of values that can be communicated to the values instantiated in the Read and Show classes.

```haskell
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 (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. 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 MChannel is presented. First 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`. 
1.2.2 Discovering Resources

One of the main objectives for using mobile computation is to better explore the resources available in a network. Hence, if a program migrates from one node of the network to another, this program must in some way find the resources available in the destiny.

type ResName = String

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

In figure 1.4, we present three primitives for resource discovery and registration. All machines running mHaskell 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:

\[
\text{rfork :: IO ()} \to \text{HostName} \to \text{IO ()}
\]

It takes an IO action as an argument and sends it to be evaluated in the remote host $\text{HostName}$. The $\text{rfork}$ function is implemented using MChannels as described in [BTL03].

1.3 IMPLEMENTATION DESIGN

1.3.1 Introduction

Mobile systems have to 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 a 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 RTS, based on the STG-machine [Jon92], that is a graph reduction machine.

This section explains how the implementation of $m$Haskell using the GHC compiler works. First we discuss some design options in the language level such as the evaluation of thunks and how to deal with shared graphs. Then we discuss how the thunks are evaluated. Finally, we explain the packing routines and the implementation of MChannels.

1.3.2 Communication of Thunks

When a value is sent through a channel, it is evaluated to normal form before communication occurs. The reason for that is because Lazy Evaluation imposes some complications for reasoning about the communication of mobile programs. Consider the following example:

\[
\text{let }
(a, b, c) = f x \\
\text{in }
\text{if } a \text{ then }
\text{writeMChannel ch b}
\]

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. When we evaluate \( a \) in the \( \text{if} \) statement, the function application \( f \ x \) will also be evaluated and the graph for \( b \) will be similar to the one in figure 1.5.

\[
\text{FIGURE 1.5. Graph for } b
\]

At the point where \texttt{writeMChannel} is performed, the value \( b \) is represented in the heap as the \texttt{selector} that gets the second value of a tuple applied to the whole tuple. If \texttt{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 won't be affected by an evaluation using the \texttt{seq} function, and this evaluation to normal form won't affect programs using \texttt{rfork} because all computations started using remote thread creation 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. Thunks can also be sent if we wrap them in an IO value, like:

\[
\begin{align*}
\text{apply} \ (a->b) \ &\to \ a \ -> \ \text{IO} \ b \\
\text{apply} \ f \ x \ &= \ \text{return} \ (f \ x)
\end{align*}
\]

1.3.3 Sharing Properties

Modern non-strict functional languages usually have their implementation based on a graph reduction machine where a program is represented as a graph and the evaluation of the program is performed by reducing the graph. Programs are represented as graphs to ensure that shared expressions are evaluated only once.

If we try to maintain sharing between nodes in our distributed system this 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 this structures) that the programmer of the system did not know about. 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 notebook). If we preserve sharing, 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 also partially solved by making the primitives strict because then expressions will be evaluated only once and only the result of the evaluation is communicated.

1.3.4 Evaluating Thunks

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

```haskell
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:

```haskell
f :: a -> b -> Int
let
  a = (...)
in
writeMChannel ch (f a)
```

In this case it is not possible, inside of the definition of `writeMChannel`, to evaluate the thunk for `a` using strategies.

One solution to this problem would be to implement a function `kids` with type:

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

It takes a value from the heap as its argument and returns 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 system: using a primitive function that creates a new RTS thread to evaluate its argument.

In our implementation we use an hybrid approach: a thunk in the top level of the graph for the computation that is going to move is forced by a `seq` (as in figure 1.6). If there are other thunks in the graph, this thunks are evaluated with an extra-thread in the RTS. The problem with this approach is that while a thunk is being evaluated by this thread, the garbage collector might be started and that will destroy the queue with closures yet to be packed. One solution to this problem is to make this queue visible to the Garbage Collector (the queue will be a source of roots for GC) then after GC we still have valid pointers at the queue.

1.3.5 Serialising Graph

We have implemented routines for packing and unpacking the graph structures that represent the computation being communicated. These routines are based on
the GUM [THM+96] system but extended to pack GHCi’s Byte-Code Objects (BCOs). By packing we mean *serialising* these objects into a form that is suitable for communication.

Packing arbitrary graph is not a trivial task and care must be taken in order to preserve sharing and cycles [THM+96]. 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 in order 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 bytecode. 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 bytecode instructions thus all this closures 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 1.3.6.

As the basic modules in 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 definitions 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 bytecode. Once packed, the BCOs can be communicated in the way described in section 1.3.7. 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

![Diagram of thunks evaluation using `seq`]

**FIGURE 1.6.** Evaluation of thunks using `seq`
byte-code using GHCi.

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 both ends of the communication.

1.3.6 Communicating user defined types (ADTs)

User defined types are compiled into machine code in GHCi. There are two ways 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 destiny.

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.

1.3.7 Implementation of MChannels

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

MChannels are single reader channels. There are two main reasons for this design decision. First, because is difficult to decide to 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 this 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).

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 that it is not supposed to be receiving. This is a classic problem that appear also in the untyped \( \pi \)-calculus [Mil99].

Communication is implemented using the standard sockets library provided by the operating system, thus avoiding the need of any extra libraries e.g. PVM and 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
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 RTS occurs. When a new mobile channel (MC) 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 MC it is in fact written into its CHC. The RTS then reads this value from the CHC, serialises it and communicates it to the appropriate host based on the information present in the channel data type. When the RTS receives a value from a remote host this value is written into the CHC that represents the MC that should receive the message. A thread that reads a value from a MC 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 registerMChannel 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 in that channel.

1.4 RELATED WORK

GPh [THLP98] and Eden [BLOMP97] 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 \texttt{par} combinator in \texttt{GPH}, or process abstractions in Eden, but where and when the data will be shipped is decided by the implementation of the language.

\texttt{GdH} [PTL00], seems to be closer to the language presented here. Communication can be implemented using \texttt{MVars} and remote execution of computations is provided with the \texttt{revalIO} (remote evaluation) and \texttt{rforkIO} primitives. The problem in using \texttt{GdH} for mobile computation is that it is implemented to run on closed systems. After a \texttt{GdH} program starts running, no other PE (Processing Element) can join the computation. Another problem is that its implementation relies on a \textit{virtual shared heap} that is shared by all the machines running the computation. The algorithms used to implement this kind of structure might not scale well for large distributed systems like the Internet.

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 \texttt{Show} class. Furthermore, the types of the messages which can be received must be an instance of the \texttt{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 in the \texttt{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. The problem of their approach is that they do not have a real implementation of concurrency, their system provides interleaved execution of atomic actions implemented using a continuation monad.

1.5 CONCLUSIONS AND FUTURE WORK

We have presented the implementation of \texttt{mHaskell}, an extension of Haskell for the implementation of distributed mobile systems.

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.

• A cache for the functions that were already communicated.

• Some languages that support mobility of code also support the migration of running computations. We could also extend Haskell with a primitive for transparent strong mobility, that would be a primitive to explicitly migrate threads:

\[
moveTo :: \text{HostName} \rightarrow \text{IO}()
\]

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 on 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 helpfully comments on the implementation design for `mHaskell`. Bernard Pope also gave important suggestions about the evaluation of thunks. This work has been partly supported by an ORS Scholarship.

**REFERENCES**


