*m*Haskell: Mobile Computation in a Purely Functional Language

André Rauber Du Bois^{1,2} **, Phil Trinder**^{1} **, Hans-Wolfgang Loidl**^{3}

^{1} School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.

^{2}Escola de Informática,
Universidade Católica de Pelotas, CEP: 96010-000 , Pelotas-RS, Brazil

^{3}Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany

{dubois,trinder}@macs.hw.ac.uk,hwloidl@informatik.uni-muenchen.de

Abstract. *This paper is a complete description of mHaskell, an extension of Concurrent Haskell for mobile computation. We describe new stateful mobility primitives that use higher-order channels, giving their operational semantics and an implementation outline. We show how medium-level coordination abstractions can be constructed using monadic composition of the mobility primitives. We briefly outline how high-level mobile coordination abstractions, or* mobility skeletons, *can be defined using the lower-level abstractions. The use of all three abstractions is demonstrated with examples and a new case study: a distributed stateless web server where a thread farm skeleton is used to distribute work to remote locations.*

1. Introduction

Networks are increasingly pervasive, and *mobile computation* languages are one means of exploiting them [Fuggetta et al., 1998]. Mobile languages enable the programmer to control the placement of active computations, and execute on open networks, i.e., a network where locations, or machines, can dynamically join and leave the computation. A mobile program can transport its state and code to another location in the network, where it resumes execution [Lange and Oshima, 1999].

In a functional language functions are first class values, and in a distributed language we expect to be able to communicate functions and computations. A language that can communicate functions in an open system can be seen as a mobile computation language, or mobile language. Many mobile languages are functional, e.g., Jocaml [Conchon and Fessant, 1999], Nomadic Pict [Wojciechowski, 2000], Kali Scheme [Cejtin et al., 1995], Facile [Knabe, 1995].

Mobile languages are fundamentally different from parallel languages. The objective of a mobile program is usually to exploit the resources available at specific locations, e.g., databases, programs, or specific hardware. As a result mobile programs are usually stateful and in a purely functional context the stateful components must be carefully managed to preserve referential transparency, e.g., using monads in Haskell. In contrast parallel functional languages can freely distribute stateless computations.

This paper presents *m*Haskell, a mobile language based on the Haskell purely functional language. Haskell uses *monads* to specify stateful computations: stateful operations are encapsulated in an abstract I/O action data type [Peyton Jones and Wadler, 1993]. Haskell computations are first-class values, i.e., functions can receive actions as arguments, return actions as results, and actions can be combined to generate new actions. The crucial implication for a mobile language is that computations can be manipulated and new abstractions over computations defined.

The paper starts by describing the new higher-order communication primitives (Section 2). We show how they can be used to implement more powerful abstractions such as remote thread creation (Section 3). Examples of using the primitives and other abstractions are given (Section 4), as is a distributed stateless web server case study (Section 5). The *m*Haskell operational semantics is outlined (Section 6), and related work described (Section 7).

This paper is the first complete description of *m*Haskell together with its semantics. Earlier papers have described the implementation [Du Bois et al., 2004a], and the definition of mobility skeletons, or polymorphic high-level mobile coordination abstractions, with a distributed meeting scheduler case study [Du Bois et al., 2004b]. Additional contributions of this paper are a new and unusual case study: the distributed stateless webserver, implemented using a new interface to the serialisation routines, and the design, implementation and use of a new mobility skeleton: a thread farm.

2. Mobile Haskell

2.1. MChannels

Mobile Haskell or *m*Haskell is a small conservative extension of Concurrent Haskell [Peyton Jones, 2001]. It enables the construction of distributed mobile software by introducing higher order communication channels called *Mobile Channels*, or MChannels. MChannels allow the communication of arbitrary Haskell values including functions, IO actions and channels. Figure 1 shows the MChannel primitives.

```
data MChannel a      -- abstract data type
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: Mobile Channels

The exact meaning of these primitives is defined in the operational semantics in Section 6, here we give an informal explanation. 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 synchronous, when a `readMChannel` is performed in an empty MChannel it will block until a value is received on that MChannel. When used locally, MChannels have similar semantics to Concurrent Haskell channels. In the same way, when a value is written to a MChannel the current thread blocks until the value is received in the remote host. 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 a given name server. A name server is always running on every location of the system and a channel is always registered in the local name server with the `registerMChannel` function. MChannels are single-reader channels, meaning that only the program that created the MChannel can read values from it. Values are evaluated to normal form before being communicated, as explained in Section 2.3.

2.2. Discovering Resources

One of the objectives of mobile programming is to exploit the resources available in a dynamic network. For example, if a program migrates from one location in 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 e.g., databases, load information, local functions.

```
type ResName = String

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

Figure 2: Primitives for resource discovery

Figure 2 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 `registerRes` function takes a name (`ResName`) and a resource (of type `a`) and registers this resource with the name given. The function `unregisterRes` unregisters a resource associated with a name and `lookupRes` takes a `ResName` and returns a resource registered with that name in the *local* registration service. To avoid a type clash, an abstract type must be defined to hold the different values that can be registered.

Resources with differing types can be elegantly handled using dynamic types. The *Glasgow Haskell Compiler* (GHC) [GHC, 2005] supports a simple form of dynamic types [Lämmel and Peyton-Jones, 2003], providing operations for injecting values of arbitrary types into a dynamically typed value, and operations for converting dynamic values into a monomorphic type.

2.3. Strictness and Sharing

To simplify *m*Haskell's semantics, values are evaluated to normal form before being sent through an MChannel.

*m*Haskell evaluates all thunks (unevaluated expressions) in the graph that is being communicated. The evaluation of thunks affects only pure expressions, as it is difficult to predict the amount of graph that is being communicated in a lazy language. IO computations are not executed during this evaluation step, and most mobile programs are stateful IO actions that interact with remote resources.

Haskell, like other non-strict languages, is commonly implemented using graph reduction which ensures that shared expressions are evaluated at most once. Maintaining sharing between graph nodes in a distributed system would result in a generally 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). In *m*Haskell, computations are *copied* between machines and no sharing is preserved *across* machines, although sharing is preserved in the value being communicated.

2.4. Implementation

Mobile languages require an architecture neutral code representation and *m*Haskell has been implemented as an extension to the GHC compiler, which supports both byte-code and native code. GHC combines an optimising compiler and the GHCi interactive environment. 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.

To implement *m*Haskell, GHCi's runtime system was extended with routines to *serialise* Haskell expressions, i.e., convert expressions into an array of bytes that can be easilly communicated. A description of the low-level details of the implementation, e.g., graph packing/unpacking, and distributed communication can be found in [Du Bois et al., 2004a]. Here we extend the implementation by giving a simple interface to these routines through two new Haskell primitives, `packV` and `unpackV`:

```
packV :: a -> IO CString
unpackV :: CString -> IO a
```

The `packV` primitive takes a Haskell expression and returns a C array with the expression serialised, and `unpackV` converts the array back into a Haskell value.

Here is a simple example showing the use of the primitives:

```
main = do
    buff <- packV plusone
    newplusone <- unpackV buff
    print ("result " ++
         show ((newplusone::Int->Int) 1 ))
    where
     plusone ::  Int ->Int
     plusone x = x + 1
```

These primitives, as described in the case study, can be used to implement other extensions to the compiler, e.g., a library for persistent storage of programs. Again, dynamic types could be used to ensure that the computations, when executed, have the right type.

A key issue in a mobile language is to control how much code moves, e.g., should primitives for addition and printing be copied? If the complete representation of the computation is not communicated, the mobile code must dynamically link to code at the destination location. Mobile Haskell provides a simple means of controlling mobility: only modules compiled to byte code are communicated. Modules like the prelude that are compiled to machine code are not communicated, and are dynamically linked at the destination.

3. Mobility Abstractions

3.1. Mid-Level Abstractions

Using MChannels the programmer can specify low level coordination details, e.g., communication and synchronisation of computations. In this section we add another layer of abstraction to *m*Haskell by showing how remote thread creation can be implemented using MChannels. Every location in the system should run a remote fork server (`startRFork`) (Figure 3). The server creates a MChannel with the name of the location

```
startRFork = do
  mch <- newMChannel
  name <- fullHostName
  registerMChannel mch name
  rforkServer mch
   where
   rforkServer mch = do
   comp <- readMChannel mch
   forkIO comp
   rforkServer mch
```

```
rfork:: IO () -> HostName -> IO()
rfork io host = do
 ch <- lookupMChannel host host
 case ch of
  Just nmc -> writeMChannel nmc io
  Nothing  -> error "Remote Server
                    unavailable"
```

Figure 3: Implementation of `rfork`

in which it is running. After that, it keeps reading values from the MChannel and forking threads with the IO actions received. Threads are forked using the `forkIO` function of Concurrent Haskell. If all locations in the system are running this server, the `rfork` function is easily implemented. The `rfork` function looks for the channel registered in the `startRFork` server, and sends the computation to be forked on the remote location `host`.

A computation can be sent to be evaluated on a remote location using the `reval` function, which is easily implemented using `rfork`:

```
reval :: IO a -> HostName -> IO a
```

3.2. Mobility Skeletons

Some of the main advantages of functional languages are the ease of composing computations and the powerful abstraction mechanisms. In separate work [Du Bois et al., 2004b], we have used these techniques to develop parameterisable higher-order functions in *m*Haskell that encapsulate common patterns of mobile computation, so called *mobility skeletons*. These skeletons hide the coordination structure of the code, analogous to algorithmic skeletons [Cole, 1989] for parallelism. Mobility skeletons abstract over mobile stateful computations on open distributed networks. In Section 5, we identify and implement a new skeleton, the `threadFarm`.

4. Mobility Example: Determining Remote Load

Figure 4 shows a *m*Haskell program that measures the load on a remote location. It starts by creating a new MChannel for the result. Then it forks a remote thread to execute `code`, and waits for its reply by executing `readMChannel`. When the thread `code` is spawned on a remote machine (in the example `ushas`), it obtains its load using a local function called `getLoad`, that was previously registered in the resource server:

```
   registerRes getLoad "getLoad"
```

After executing `getLoad`, it sends the result back to the main program where it is printed.

The previous program can be extended to calculate the total load of the network, as in Figure 5.

In this example, instead of using a channel to return the result of the remote computation, `reval` is used. The `code` function, is modified to return a `Maybe` value, so it is possible to detect failures in finding the `getLoad` resource on the remote locations.

```
main = do
   mch <- newMChannel
   rfork (code mch) "ushas.hw.ac.uk"
   load <- readMChannel mch
   print ("Load on ushas: "++ (show load))
   where
     code :: MChannel Int -> IO ()
     code mch = do
       func <- lookupRes  "getLoad"
       case func of
        Just gL   -> do
                load <-gL
                writeMChannel mch load
         _          -> recoverError

     recoverError = (...)
```

Figure 4: Sending code **to** ushas

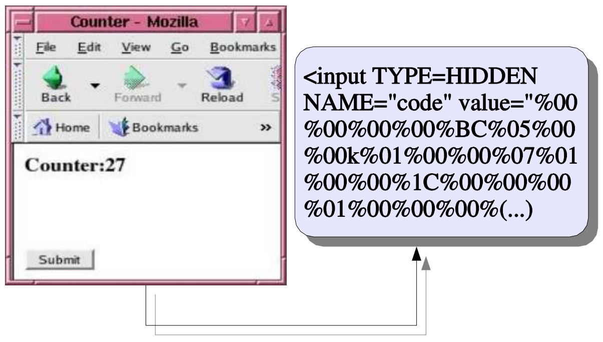
```
main = do
   load <- mapM (reval code) listofmachines
   print load
   where
   code :: IO (Maybe Int)
   code = do
       func <- lookupRes  "getLoad"
       case func of
        Just gL   -> do
                load <-gL
                return (Just load)
        Nothing   -> return Nothing
```

Figure 5: Finding the load of a network

5. Case Study

5.1. Stateless Servers

In stateless servers [Halls, 1997], all persistent knowledge about applications is kept in the documents exchanged between clients and servers. Servers and clients are stateless:



Counter:27

<input TYPE=HIDDEN NAME="code" value="%00%00%00%00%BC%05%00%00k%01%00%00%07%01%00%00%1C%00%00%00%01%00%00%00%(...)

Figure 6: A simple counter page

the server stores the continuation of its application in the document that it returns to the client. When it receives a new request from a client, it doesn't need to remember anything from the previous interaction, it simply executes the continuation contained in the new input.

Mobile languages are a perfect platform for the implementation of stateless servers because running applications can be saved, communicated and restarted. In this section, we use the serialisation primitives, described in Section 2.4, to implement, following the ideas presented in [Halls, 1997], a stateless web server that keeps the state of its computations in the pages that it sends to browsers, and executes code sent in the clients requests. Furthermore, by using MChannels and a thread farm, we make the web server distributed, using a cluster of machines to process the requests sent by clients.

As in [Marlow, 2000], the web server is implemented by using a simple main loop, that repeatedly reads requests from a socket and forks threads to process these requests. The main difference occurs when one of the messages contains the code for an application:

```
(codeBuffer,args) <- getCode clientMsg
computation <- unpackV codeBuffer
responseToClient <- computation args
```

In this case, we use the `getCode` function, that processes the string representing the client request, separating the serialised code in it from the normal arguments in a POST message. The code is unpacked into the heap using the `unpackV` function, and executed. The computation should generate a new web page with results to be sent back to clients. If further client interaction is required, the web page generated will contain a continuation.

For simplicity we assume that the computation stored in the client has type `[(String,String)] -> IO String`, where the argument string has the values sent by the client in the POST message. To avoid any type clashes, Haskell's dynamic types could be used to ensure that the computation has the right type.

5.2. A Counter

As an example, we present the implementation of a simple counter web page where the server increments the counter and saves its continuation in the page, as illustrated in Figure 6. The counter is implemented as follows:

```
counter :: Int -> [(String,String)] -> IO String
counter n _ = do
      cs <- counterPage (n+1) (counter (n+1))
      return cs
```

It takes as an argument its current state (an `Int`) and uses the `counterPage` function to generate the response that is sent back to web clients. The `counterPage` action generates a new web page (a `String`), that displays the current state of the counter in HTML (its first argument), and uses the `packV` function to serialise its second argument, that is, the continuation of the computation. The `counter` just ignores its second argument, the contents of the POST message.

Every time that a browser sends a request for the root document (/) the counter is started with zero:

```
str <- counter 0 emptyArg
sendResp socket str
```

```
threadFarmServer ::
        IO (MChannel (Maybe (IO ())), MChannel HostName)
threadFarmServer = do
   ioc <- newMChannel
   hostc <- newMChannel
   forkIO (serverth ioc hostc)
   return (ioc,hostc)
   where
    serverth ioc hostc = do
       v <- readMChannel ioc
       case v of
        Just action -> do
           host <- readMChannel hostc
           forkIO (handleCon action hostc host)
           serverth ioc hostc
        Nothing     -> return ()
    handleCon action hostc host= do
       empty <- reval action host
       writeMChannel hostc host
```

Figure 7: The thread farm server

and the page generated by the counter, containing its continuation, is sent back to the client.

When the user presses the submit button in the web page (Figure 6), and the POST message arrives in the web server, the continuation is unpacked, run, and a new continuation is sent back to the client.

5.3. A Distributed Web Server

A web server may be overloaded if it receives a huge amount of requests from clients asking to execute computations. In a mobile language it is possible to offload a server by sending computations to be executed on other locations. With *m*Haskell, it is easy to make the web server distributed: a cluster of machines can be used to execute the computations sent by clients. A *thread farm* can be used in order to provide round-robin scheduling of the tasks in the machines available for processing.

The thread farm is implemented using a server that reads actions from an MChannel, and sends these actions to be executed on remote machines that it gets from another MChannel (Figure 7). The `threadFarmServer`, when started, returns two MChannels, that can be used to dynamically increase the number of computations and machines used in the system. After creating the two MChannels the main thread of the server, `serverth`, is forked. The server thread reads `Maybe (IO())` values from the `ioc` MChannel. The main thread is stopped once it reads a `Nothing` from `ioc`. When the thread finds a value in the MChannel, it gets one remote machine from `hostc` and starts another thread to handle the execution of the remote computation. The `handleCon` function starts the remote execution of `action` on `host` and, after it completes, `host` is returned to the MChannel of free machines. Remote evaluation (`reval`) is used in this case, instead of `rfork`, because we want to be sure that the remote machine being used in the computation is only returned to the list of free machines once the remote computation has been completed.

The `threadFarm` function can be implemented as in Figure 8. It takes as an argument a list of actions to be executed on remote locations, and a list of locations. It

```
threadFarm :: [IO ()] -> [HostName] ->
                       IO (MChannel (Maybe (IO ())))
threadFarm comp names = do
   (ioc,hostc) <- threadFarmServer
   mapM_ ( writeMChannel hostc) names
   mapM_ ( writeMChannel ioc . Just ) comp
   return ioc
```

Figure 8: The thread farm

returns an MChannel that can be used to send more computations to the thread farm, or to stop the thread farm server by writing a `Nothing` in it. The `threadFarm` starts the server and then writes the initial values and locations into their respective channels.

In the case of the web server, the `threadFarm` can be started with an empty list of computations, and the `tfMChannel` returned by `threadFarm` is used to send the computations received from clients to the remote thread server:

```
 tfMChannel <- threadFarm [] listOfMachines
```

Computations executed by the `threadFarm` must have type `IO ()`, but in the case of the web server, the computations received by clients, after given their argument, have type `IO String`. Furthermore, the web server wants to receive the result of the computation (the `String` with the page that must be sent to clients) back from the remote location that executed the computation. This is achieved by wrapping the computation in an IO action that executes the computation and sends its result back through a MChannel, as can be seen in this modified version of the program to handle POST messages:

```
(...)
 resp <- newMChannel
 writeMChannel tfMChannel (Just (execComp resp (computation args))
 string <-readMChannel resp
 sendResp socket string
 where
   execComp :: MChannel String -> IO String -> IO ()
   execComp ch comp =do
      str <- comp
      writeMChannel r str
```

The computation is sent to the thread farm through the `tfMChannel`, and it is wrapped in the `execComp` action, that just executes its argument, and sends its result back to the web server through a channel. Exactly the same approach is used to implement remote evaluation in terms of `rfork`.

The thread farm implemented in *m*Haskell is different than a parallel skeleton because the code for the computations does not need to be present in the remote locations, and new locations can be added dynamically to the thread farm.

6. Operational Semantics

This section gives the operational semantics for MChannels by extending the semantics for monadic IO presented in [Peyton Jones, 2001]. The semantics has two levels: an *inner denotational semantics* for pure terms that is standard and not described here, but could be based, for example, on [Moran et al., 1999]; and an *outer transitional semantics* describing the IO actions and MChannels. Our extensions only affect the transitional semantics.

	con	$\in$	*Constructor*	
	ch	$\in$	*Char*	
	x	$\in$	*Variable*	
	t	$\in$	*ThreadId*	
Value	V	$::=$	$\backslash x- > M \mid conM_1 \cdots M_n \mid ch$	
			$\mid$ `return` $M \mid M$ `>>=` N	
			$\mid$ `putChar` $ch \mid$ `getChar`	
Term	M, N	$::=$	$V \mid M\ N \mid$ `if` M `then` N_1 `else` $N_2 \mid \cdots$	
SState	P, Q, R	$::=$	$\{M\}_t$	A thread called t
		$\mid$	$P \mid Q$	Parallel Composition
		$\mid$	$\nu x.P$	Restriction
Evaluation contexts	$\mathbb{E}$	$::=$	$[\cdot] \mid \mathbb{E}$ `>>=` M	

Figure 9: Syntactic and semantic domains of the basic functional language

Figure 9 gives the syntax of a simple Haskell-like functional language. Describing the syntactic domains, M and N range over *Terms* and V over *Values*. A value is something that is considered by the inner, purely-functional semantics as evaluated. Primitive monadic `IO` operations are treated as values, e.g., `putChar 'c'` is a value as no further work can be done on this term in the purely-functional world. Variables P, Q and R range over elements of the single-processor state *SState* and can be threads, MChannels (defined in the extended syntax in Figure 11), a composition of two states using the $\mid$ combinator, or a restriction of a program state over a variable. A thread is an active element where the evaluation occurs, while MChannels, like MVars [Peyton Jones, 2001], are just containers for values that are used during the evaluation.

To identify the next transition rule to be applied, *evaluation contexts* are used. An evaluation context $\mathbb{E}$ is a term with a hole $[\cdot]$, and its syntax is presented in Figure 9. The symbol $[\cdot]$ indicates the location of the hole in an expression and $\mathbb{E}[M]$ is used to show that the hole $\mathbb{E}$ is being filled by the term M. The basic transition rules for simple monadic actions are presented in Figure 10. The transition from one program state to the next may or may not be *labelled* by an *event*, α, representing communication with the external environment, e.g., input (**?**) or output (**!**). For a detailed description of the semantics of monadic actions, the reader should refer to [Peyton Jones, 2001].

$$\{\mathbb{E}[\texttt{putChar}\ ch\ ]\}_t \xrightarrow{!ch} \{\mathbb{E}[\texttt{return}\ ()]\}_t \quad (\textit{PUTC})$$

$$\{\mathbb{E}[\texttt{getChar}\ ]\}_t \xrightarrow{?ch} \{\mathbb{E}[\texttt{return}\ \ ch]\}_t \quad (\textit{GETC})$$

$$\{\mathbb{E}[\texttt{return}\ \ N >>= M]\}_t \longrightarrow \{\mathbb{E}[M\ N]\}_t \quad (\textit{LUNIT})$$

$$\frac{\varepsilon[[M]] = V \quad M \not\equiv V}{\{\mathbb{E}[M]\}_t \rightarrow \{\mathbb{E}[V]\}_t} \quad (\textit{FUN})$$

Figure 10: Basic transition rules

In Figure 11, we extend the syntactic and semantic domains for IO actions with

MChannels. New variables are added to represent MChannel identifiers (c), location names (s), MChannel names (n), and remote references to MChannels (r). The new primitives on MChannels are added to the syntactic domain of the language. The semantic domain is augmented with a data structure representing MChannels, which is recursively defined and uses list-like operations for adding an element to the front and appending an element to the end. We don't give a formal definition of this data structure here, but observe that it is used as a queue. The overall state L is defined as a finite map from location names (s) to single-processor states (P). Thus, $L(s) = \{M\}_t$ describes that M is being executed at location s. We write $L(s, P)$ to indicate that the finite map L is extended with the binding $s \mapsto P$, shadowing any previously binding of s in L.

	c	$\in$	$MChannel$	MChannel identifi er
	s	$\in$	$LName$	Names of locations in the distributed system
	n	$\in$	$MName$	MChannel names
	ε	$\in$	$MName$	an empty MChannel name
	$r_{n@s}$	$\in$	$RRef$	Remote Reference to MChannel n at location s
Value	V	$::=$	... | `newMChannel` | `writeMChannel` $c\ M$	
		|	`readMChannel` c | `registerMChannel` $c\ n$	
		|	`unregisterMChannel` n | `lookupMChannel` $s\ n$ | s | n | ε	
SState	P, Q, R	$::=$	...	
		|	C_n^c	An `MChannel` c with name n
State: Exp	L	$::=$	$LName \to SState$	
Mchannel	C_n^c	$::=$	$\langle\rangle_n^c$	An empty `MChannel` c with name n
		|	$M : C_n^c$	A `MChannel` c with head value M and tail C
		|	$C_n^c \ {+\!\!+} \langle M\rangle_n^c$	A `MChannel` c with M as the last element

Figure 11: Extended syntactic and semantic domains of the language with MChannels

The transition rules for MChannels are presented in Figure 12. Common congruence rules, such as commutativity and associativity of $|$, follow from the observation that the semantics for Concurrent Haskell [Peyton Jones, 2001] is a special case of our semantics for just one processor. Rule (*NEWC*) creates a new state with an empty MChannel that has no name (ε), and executes in parallel with the current thread. The (*REGC*) and (*UNREGC*) rules set and unset the name n of a MChannel. The `readMChannel` primitive, reads a term M from a local MChannel. It returns the first element in the structure, and cannot be applied to a remote reference, reflecting the fact that the MChannels are single reader channels: only threads running on the location where the MChannel was created can read from it. The `lookupMChannel` function, returns a remote reference $r_{n@s'}$ to a remote MChannel at location s', if it exists. Otherwise it should return the value `Nothing`, but this is left out of the semantics for simplicity. The `writeMChannel` primitive can be applied to the identifier of a MChannel that runs on the same location or to a remote reference. Rule (*WRITECl*) specifies that writing to a local MChannel appends the value, M, as the last element. Rule (*WRITECr*) states that if `writeMChannel` is executed at location s, and is applied to a reference $r_{n@s'}$ to a channel located at s', the term M is sent to location s' and written into the channel. Before M is communicated the function `forceThunks` is applied to it which forces the evaluation of pure expressions (*thunks*) in the graph of the expression M, without executing the monadic actions, and

$$\frac{c \notin fn\,(\mathbb{E})}{L(s,\{\mathbb{E}[\texttt{newMChannel}]\}_t) \rightarrow L(s,\nu c.(\{\mathbb{E}[\texttt{return}\ c]\}_t) \mid \langle\rangle^c_\varepsilon))} \qquad (\mathit{NEWC})$$

$$L(s,\{\mathbb{E}[\texttt{registerMChannel}\ c\ n]\}_t \mid C^c_\varepsilon) \rightarrow L(s,\{\mathbb{E}[\texttt{return}\ ()]\}_t) \mid C^c_n) \qquad (\mathit{REGC})$$

$$L(s,\{\mathbb{E}[\texttt{unregisterMChannel}\ c]\}_t \mid C^c_n) \rightarrow L(s,\{\mathbb{E}[\texttt{return}\ ()]\}_t) \mid C^c_\varepsilon) \qquad (\mathit{UNREGC})$$

$$L(s,\{\mathbb{E}[\texttt{readMChannel}\ c]\}_t \mid M : C^c_n) \rightarrow L(s,\{\mathbb{E}[\texttt{return}\ M]\}_t) \mid C^c_n) \qquad (\mathit{READC})$$

$$\frac{r \notin fn\,(\mathbb{E}) \qquad L(s',\{\mathbb{E}[M']\}_{t'} \mid C^c_n)}{L(s,\{\mathbb{E}[\texttt{lookupMChannel}\ s'\ n]\}_t) \rightarrow L(s,\nu r.(\{\mathbb{E}[\texttt{return}\ r_{n@s'}]\}_t)\ )} \qquad (\mathit{LOOKUPC})$$

$$L(s,\{\mathbb{E}[\texttt{writeMChannel}\ c\ M]\}_t \mid C^c_n) \rightarrow L(s,\{\mathbb{E}[\texttt{return}\ ()]\}_t) \mid C^c_n +\!\!+ \langle M\rangle^c_n) \qquad (\mathit{WRITECl})$$

$$\begin{aligned} L(s,\{\mathbb{E}[\texttt{writeMChannel}\ r_{n@s'}\ M]\}_t)\ (s',\{\mathbb{E}[M']\}_{t'} \mid C^c_n) \rightarrow \\ L(s,\{\mathbb{E}[\texttt{return}\ ()]\}_t)\ (s',\{\mathbb{E}[M']\}_{t'} \mid C^c_n +\!\!+ \langle V'\rangle^c_n) \\ \text{where } V' = \texttt{forceThunks}\ M \end{aligned} \qquad (\mathit{WRITECr})$$

Figure 12: Transition rules for the language with MChannels

substitutes every occurrence of a MChannel in the graph by a remote reference.

The function `forceThunks` is not formalised here, although it can be defined in the semantics of a language with an explicit heap with thunks such as [Tolmach and Antoy, 2003].

Figure 13 shows the evaluation of the following *m*Haskell program using the semantics:

```
server =newMChannel >>= \mch ->
        registerMChannel mch n >>
        readMChannel mch >>= \io ->
        io

client =lookupMChannel s’ n>>=\mch ->
    writeMChannel mch hello
     where
      hello = print "Hello "++"World"
```

There are two important things to notice in the evaluation. Firstly, the non-determinism of the semantics: as in a real system, if the `client` looks for a channel before the `server` registers it, the program fails at the the (*LOOKUPC*) rule. Secondly in the evaluation of thunks, the strings `"Hello "` and `"World"` are concatenated before communication occurs, but IO actions are not evaluated by `forceThunks`, hence evaluation of `print "Hello world"` occurs only on the server s'.

7. Related Work

There are numerous parallel and distributed Haskell extensions, and only those closely related to *m*Haskell are discussed here. Of these languages, GDH [Pointon et al., 2000] is the closest to *m*Haskell. The problem in using GDH for mobile computation is that it is implemented to run on closed systems, that is, after a GDH program starts running, no processors can neither join nor leave the computation. Haskell with ports [Huch and Norbisrath, 2000] has primitives for communication very similar to the ones presented here, and supports distribution on open systems. The only drawback is

$$\begin{array}{lll}
 & \texttt{server}: & \\
 & L(s', \{\texttt{newMC} \;>>=\; \backslash\texttt{mch}- > \texttt{regMC mch}\; n>> \texttt{readMC mch} >>= \backslash\texttt{io}- > \texttt{io}\}_t) & \\
\rightarrow & L(s',\ \nu c.(\{\texttt{return}\ c >>= \ \backslash\texttt{mch}- > \texttt{regMC mch}\ \ n>> \texttt{readMC mch} >>= \backslash\texttt{io}- > \texttt{io}\}_t \mid \langle\rangle^c_\varepsilon)) & (\textit{NEWC}) \\
\rightarrow & L(s',\ \nu c.(\{(\backslash\texttt{mch}- > \texttt{regMC mch}\ \ n>> \texttt{readMC mch} >>= \backslash\texttt{io}- > \texttt{io})\ \texttt{c}\}_t \mid \langle\rangle^c_\varepsilon)) & (\textit{LUNIT}) \\
\rightarrow & L(s',\ \nu c.(\{\texttt{regMC}\ \ c\ \ n>> \texttt{readMC}\ c>>= \backslash\texttt{io}- > \texttt{io}\}_t \mid \langle\rangle^c_\varepsilon)) & (\textit{FUN}) \\
\rightarrow & L(s',\ \nu c.(\{\texttt{readMC}\ c>>= \backslash\texttt{io}- > \texttt{io}\}_t \mid \langle\rangle^c_n)) & (\textit{REGC})
\end{array}$$

$$\begin{array}{lll}
 & \texttt{client}: & \\
 & L(s, \{\texttt{lookup}\ s'\ n >>= \ \backslash\texttt{mch}- > \texttt{writeMC mch hello}\}_t) & \\
\rightarrow & L(s,\ \nu r.(\{\texttt{return}\ r_{n@s'} >>= \ \backslash\texttt{mch}- > \texttt{writeMC mch hello}\}_t)) & (\textit{LOOKUPC}) \\
\rightarrow & L(s,\ \nu r.(\{(\backslash\texttt{mch}- > \texttt{writeMC mch hello})\ \ \texttt{r}_{\texttt{n@s}'}\}_t)) & (\textit{LUNIT}) \\
\rightarrow & L(s,\ \nu r.(\{\texttt{writeMC}\ r_{n@s'}\ \texttt{hello}\}_t)) & (\textit{FUN})
\end{array}$$

$$\begin{array}{lll}
 & \texttt{server}: & \\
\rightarrow & L(s',\ \nu c.(\{\texttt{readMC}\ c>>= \backslash\texttt{io}- > \texttt{io}\}_t \mid \langle\texttt{print ``Hello World''}\rangle^c_n)) & (\textit{WRITECr}) \\
\rightarrow & L(s',\ \nu c.(\{\texttt{return(print ``Hello World'')} >>= \backslash\texttt{io}- > \texttt{io}\}_t \mid \langle\rangle^c_n)) & (\textit{READC}) \\
\rightarrow & L(s',\ \nu c.(\{(\backslash\texttt{io}- > \texttt{io})\ \ (\texttt{print ``Hello World''})\}_t \mid \langle\rangle^c_n)) & (\textit{LUNIT}) \\
\rightarrow & L(s',\ \nu c.(\{(\texttt{print ``Hello World''}\}_t \mid \langle\rangle^c_n)) & (\textit{FUN}) \\
\rightarrow & L(s',\ \nu c.(\{(\texttt{return ()}\}_t \mid \langle\rangle^c_n)) & (\textit{PRINT})
\end{array}$$

Figure 13: Example of evaluation using the semantics

that the current implementation of the language only supports communication of first-order values: functions and IO actions cannot be communicated. Another closely related system is Famke [van Weelden and Plasmeijer, 2002], an implementation of threads for the lazy functional language Clean 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 continuation monad.

There are other extensions to functional languages that allow the communication of higher-order values. Kali-Scheme [Cejtin et al., 1995] is an example of a strict weakly typed language that allows the communication of functions. Other strict typed languages such as Nomadic Pict [Wojciechowski, 2000], Facile [Knabe, 1995] and Jocaml [Conchon and Fessant, 1999] implement the communication primitives as side effects while we integrate them to the IO monad, preserving referential transparency.

8. Conclusions

The *m*Haskell mobile language has been described, covering its semantics, the definition of a range of coordination abstractions, its implementation, and a case study. *m*Haskell differs from previous extensions of Haskell in supporting higher-order communication in open distributed systems, including the communication of functions, computations and channels. *m*Haskell extends Concurrent Haskell and makes essential use of higher-order functions, polymorphism and monadic manipulation of computations. The paper extends previous *m*Haskell publications by giving a complete language description, describing a novel case study, and introducing a new mobility skeleton. In future work we plan to prove evaluation location and order properties using the operational semantics. We also plan to investigate improving the reliability and security in *m*Haskell, e.g., modelling Erlang-style behaviours, using proof carrying code [Aspinall et al., 2004] or distributed versioning of modules [Sewell, 2001].

References

Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark, I. (2004). Mobile Resource Guarantees for Smart Devices. In *CASSIS'04 — Intl. Workshop on Construc-*

tion and Analysis of Safe, Secure and Interoperable Smart Devices, LNCS, Marseille, France, March 10–13. Springer-Verlag.

Cejtin, H., Jagannathan, S., and Kelsey, R. (1995). Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5):704–739.

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman.

Conchon, S. and Fessant, F. L. (1999). Jocaml: Mobile agents for Objective-Caml. In *ASA'99/MA'99*, Palm Springs, CA, USA.

Du Bois, A. R., Trinder, P., and Loidl, H.-W. (2004a). Implementing Mobile Haskell. In *Trends in Functional Programming*, volume 4. Intellect.

Du Bois, A. R., Trinder, P., and Loidl, H.-W. (2004b). Towards Mobility Skeletons. In *CMPP'04 — Constructive Methods for Parallel Programming*, Stirling, Scotland.

Fuggetta, A., Picco, G., and Vigna, G. (1998). Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361.

GHC (2005). The Glasgow Haskell Compiler. WWW page, http://www.haskell.org/ghc.

Halls, D. A. (1997). *Applying Mobile Code to Distributed Systems*. PhD thesis, Computer Laboratory, University of Cambridge.

Huch, F. and Norbisrath, U. (2000). Distributed programming in Haskell with ports. In *IFL 2000*, LNCS, Volume 2011. Springer-Verlag.

Knabe, F. C. (1995). *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University.

Lämmel, R. and Peyton-Jones, S. (2003). Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of TLDI 2003*. ACM Press.

Lange, D. B. and Oshima, M. (1999). Seven good reasons for mobile agents. *Communications of the ACM*, 3(42):88–89.

Marlow, S. (2000). Writing high-performance server applications in Haskell, case study: A Haskell web server. In *Haskell Workshop*, Montreal, Canada.

Moran, A. K., Lassen, S. B., and Jones, S. L. P. (1999). Imprecise exceptions, co-inductively. In *Proceedings of HOOTS'99*, volume 26 of *ENTCS*.

Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press.

Peyton Jones, S. L. and Wadler, P. (1993). Imperative functional programming. In *Principles of Programming Languages*.

Pointon, R., Trinder, P., and Loidl, H.-W. (2000). The design and implementation of Glasgow Distributed Haskell. In *IFL 2000*, LNCS, Volume 2011. Springer-Verlag.

Sewell, P. (2001). Modules, abstract types, and distributed versioning. In *Proceedings of POPL 2001*, pages 236–247.

Tolmach, A. and Antoy, S. (2003). A monadic semantics for core Curry. In *Proc. of WFLP 2003*, Valencia (Spain).

van Weelden, A. and Plasmeijer, R. (2002). Towards a strongly typed functional operating system. In *IFL 2002*, LNCS, Volume 2670. Springer-Verlag.

Wojciechowski, P. T. (2000). *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Wolfson College, University of Cambridge.