Strong Mobility in Mobile Haskell

André Rauber Du Bois
(Escola de Informática,
Universidade Católica de Pelotas, Brazil
dubois@ucpel.tche.br)

Phil Trinder
(School of Mathematical and Computer Sciences,
Heriot-Watt University, UK
trinder@macs.hw.ac.uk)

Hans-Wolfgang Loidl
(Institut für Informatik,
Ludwig-Maximilians-Universität München, Germany
hwloidl@informatik.uni-muenchen.de)

Abstract: In a mobile language, computations can move between locations in a network to better utilise resources, e.g., as in a computational GRID. *Mobile Haskell*, or *m*Haskell, is a small extension of Concurrent Haskell that enables the construction of distributed mobile software by introducing higher order communication channels called *Mobile Channels* (MChannels). *m*Haskell only provides *weak mobility*, i.e. the ability to start new computations on remote locations. This paper shows how *strong mobility*, i.e. the ability to migrate running threads between locations, can be implemented in a language like *m*Haskell with weak mobility, higher-order channels and first-class continuations. Using Haskell's high level features, such as higher-order functions, type classes and support for monadic programming, strong mobility is achieved without any changes to the runtime system, or built-in support for continuations. Strong mobility is illustrated with examples and a mobile agent case study.
Key Words: Mobile Computation, Strong Mobility, Functional Programming, Haskell
Category: D.3.3

1 Introduction

In languages that support *mobile computation* [Cardelli 1999], executing computations can move over the network to better utilise the available resources, e.g., as in a computational GRID [Foster and Kesselman 1999]. In essence a mobile program can transport its state and code to another location in the network, where it resumes execution. Mobile computations are commonly user applications or autonomous mobile programs such as agents.

There are a number of important mobility concepts. *Hardware mobility* is the mobility of physical devices like laptops or PDAs. Software mobility, or

mobile computation, is the mobility of programs or code between locations or devices. There is a strong connection between both areas as mobile hardware can benefit from mobile computation. Mobile computation may either be weak or strong [Fuggetta et al. 1998]. *Weak mobility* is the ability to move code between locations, or basically to start new computations on remote locations. With weak mobility either a program running on a location dynamically links to the incoming code or a new thread is started to run the incoming code. *Strong mobility* is the ability to move code together with the execution state of the program, or moving *computations*. The execution is suspended, transmitted to the destination site, and resumed there. Mobile agents are a common application of strong mobility.

Languages that support weak mobility usually provide constructs for *remote evaluation*, or *remote thread creation*, which would have these types in Haskell:

```
rfork :: IO () -> HostName -> IO ()
reval  :: IO a  -> HostName -> IO a
```

The primitive for remote thread creation, `rfork`, takes as arguments a computation and a location in the network and executes the computation on that location. Remote evaluation, `reval`, executes a remote computation and returns the result of this computation to the location that called `reval`. The abstraction provided by `reval` is similar to that provided by Java's RMI [Grosso 2001], the difference being that `reval` sends the whole computation (including its code) to be evaluated on the remote host, and RMI uses proxies (stubs and skeletons) to give access to remote methods. With weak mobility, the programmer can only start remote computations and no migration of running threads is possible. Writing programs using weak mobility can be complicated in some cases, specially when the programmer wants to control where the continuation of a computation will be executed.

*m*Haskell is an extension of the purely functional language Haskell for mobile computation. It extends Haskell with higher-order polymorphic communication channels, `MChannels`. These low level constructs can be used to implement medium-level abstractions for weak mobility such as `rfork` and `reval` [Du Bois et al. 2005b].

This paper demonstrates that strong mobility can be elegantly implemented in a language with weak mobility, higher-order channels and first-class continuations. As mobile communication is stateful, the implementation also makes essential use of stateful operations, and in *m*Haskell the use of monads potentially allows reasoning even about these stateful operations. We also demonstrate the practical usability of the resulting programming style in Haskell by implementing several small examples and one realistic case study.

2 Mobile Haskell

Mobile Haskell, or *m*Haskell [Du Bois et al. 2005a], is a small 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
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 `newMChannel` function is used to create a mobile channel and the functions `writeMChannel` and `readMChannel` are used to write/read data to/from a channel. MChannels provide synchronous communication between hosts but when used locally have similar semantics to Concurrent Haskell channels. When a `readMChannel` is performed in an empty MChannel it will block until a value is received on that MChannel and, 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 the 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, but IO actions are *never* executed.

MChannels are single reader channels mainly for security: if all programs could read from a channel retrieved using `lookupMChannel`, then a client could *pretend* to be a server and steal its messages.

2.1 Finding Resources

One of the objectives of mobile programming is to better exploit the resources available in a network. Hence, if a computation migrates from one location of a network to another, this computation must be able to discover the resources available at the destination e.g. databases, local functions, load information, etc.

```
type ResName = String

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

Figure 2: Primitives for resource discovery

*m*Haskell provides primitives for resource discovery and registration (Figure 2). All locations 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 the programmer should register resources using dynamic types. The GHC [GHC 2006] Haskell compiler has basic support for 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.

2.2 Remote Thread Creation

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

```
rfork :: 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 `rfork` function has a straightforward implementation using MChannels as described in [Du Bois et al. 2005b], and the `reval` construct described in the Introduction, can be implemented in terms of `rfork`.

3   Mobile Threads

Some languages that support code mobility also support the migration of running computations or *strong mobility*. *m*Haskell could be extended with a primitive for transparent strong mobility, i.e., a primitive to explicitly migrate threads:

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

The `moveTo` primitive receives as an argument a `HostName` to where the current thread should migrate.

Strong mobility is an extension of the remote evaluation paradigm. While with `reval` the programmer can send subprograms to be executed remotely, with strong mobility, running computations migrate between hosts, hence allowing arbitrary code movement. Strong mobility is very useful when the programmer wants to control where the continuation of a computation will be executed.

Strong mobility is usually implemented in one of two ways: runtime system (RTS) support, or *continuations + weak mobility*:

- *RTS support*: In this case the language provides libraries for serialising the state of the current thread (its stack) into a stream of bytes that can be easily communicated using any network protocol (as in the Jo-caml [Conchon and Fessant 1999] system). These routines for serialisation are more complicated than those usually available in programming languages (i.e., Java), as not only data must be communicated but the state of the whole computation including registers, stacks and memory. Our work on thread migration presented in [Du Bois et al. 2002], can be seen as the first steps in providing strong mobility at the RTS level in *m*Haskell.

- *Continuations + Weak Mobility*: In languages that have native support for continuations (through a construct like `call/cc` [Friedman and Felleisen 1995]) and weak mobility, a strong mobility construct can be easily implemented by capturing the continuation of the current computation and sending it to be executed remotely (as in Kali Scheme [Cejtin et al. 1995], and Sumii's implementation of a mobile version of Scheme [Sumii 2000]). In some languages that do not have native support for continuations, strong mobility is implemented by using *code transformation*: the code of a mobile thread is transformed so that at the point where the `moveTo` construct is called, the continuation of the thread is available as an extra argument for remote execution. This approach is used in languages like *Mobile Java* [Sekiguchi 1999] and Klaim [Bettini and Nicola. 2001].

In this Section, we present a somewhat different implementation of Strong Mobility. *m*Haskell has primitives for weak mobility but the current implementations of Haskell do not have built-in support for continuations. It is well known how continuations can be elegantly implemented in Haskell using a *continuation monad* [Wadler 1995, Claessen 1999], and using Haskell's support for monadic programming and interaction between monads, a continuation monad can operate together with the IO monad, hence inheriting support for concurrent and distributed programming using Concurrent Haskell and MChannels. First, in Section 3.1 a new type of *mobile threads* based on a continuation monad is presented. Section 3.2 describes how a primitive for strong mobility can be implemented using weak mobility and the continuation monad. In Sections 3.3 to 3.5 examples of the use of mobile threads are given, including a tree search algorithm and a new implementation of the mobility skeleton `mfold` [Du Bois et al. 2005b], using `foldr` and lazy evaluation.

3.1   A Continuation Monad

To implement mobile threads, we need to have the continuation of a thread available at any time while the thread is running. The current implementations of Haskell do not have a built-in primitive to capture the continuation of a computation, as `call/cc` in the functional language Scheme [Friedman and Felleisen 1995]. To make the continuation of the current thread available in Haskell, we use a simple continuation monad, adapted from [Claessen 1999]:

```
newtype M m a = M {runC :: (a -> Action m) -> Action m}

bindC :: M m a -> (a -> M m b) -> M m b
m 'bindC' k  = M $ \c -> runC m $ (\a -> runC (k a) c)

returnC    :: a -> M m a
returnC x    = M (\c -> c x)

instance Monad m => Monad (M m) where
  m >>= k  = m 'bindC' k
  return x = returnC x
```

The `Action` data type describes what can be done in the continuation monad. The type has two values, an `Atom` that is the computation being executed, and `Stop` that is used to stop the execution of the current thread when writing escape functions. An `Atom` describes an atomic computation that when executed returns a new `Action`, which is the continuation of the current thread.

```
data Action m
  = Atom (m (Action m))
  | Stop

action    :: Monad m => M m a -> Action m
```

```
action m    = runC m (\a -> Stop)

atom       :: Monad m => m a -> M m a
atom m     =  M (\c -> Atom (do a <- m ; return (c a)))
```

The `atom` function is used to execute other monads inside the continuation monad. In Haskell, threads created using concurrent Haskell are executed inside the IO monad, hence mobile threads should be able to execute IO actions in a similar way. The `atom` function, can be used to transform any IO action (of type `IO a`) into a *mobile action* of type `M IO a`. The monad `M` is a monad transformer, and the act of transforming one monad into another is called lifting:

```
instance MonadTrans M where
  lift = atom
```

An `Action` can be either an `Atom` that must be executed, or a `Stop` that tells that the current computation is finished or should be aborted. A monad usually has a `run` function, that is used to execute the monad. In the case of the continuation monad, it will execute the `Actions`, until it finds a `Stop` value, meaning that the computation has finished.

```
execute :: Monad m => Action m -> m ()
execute Stop       = return ()
execute (Atom am)  = do a <- am ; execute a

run    :: Monad m => M m a -> m ()
run m  = execute (action m)
```

Mobile threads should run as real threads in the runtime system. In Concurrent Haskell, threads are forked using the `forkIO` primitive, and mobile threads are run inside of a Concurrent Haskell thread, hence providing real concurrency:

```
forkMT :: M IO () -> IO ()
forkMT io = do forkIO (run io)
               return ()
```

The `forkMT` function, takes as an argument a *mobile thread* of type `M IO ()` and creates a Concurrent Haskell thread to run the computation using the `run` function of the continuation monad.

3.2   The `moveTo` operation

The `moveTo` function, that appears in Figure 3, sends the continuation of the current thread to be executed on a remote host, and terminates the thread that called it.

It takes as an argument a `HostName` where the computation should continue its execution and uses `rfork` to start a remote thread that evaluates the continuation of the current thread. The `moveTo` operation is an escape function, meaning that the current thread finishes after sending its continuation for remote execution.

```
moveTo     :: HostName -> M IO ()
moveTo host = M (\c -> action $ lift (rfork (execute (c ())) host ))
```

Figure 3: The `moveTo` function

3.3 Example 1: Migrating a Thread

In Figure 4 a simple example using strong mobility is given. The program gets
the name of the current location, moves to a new location where the name of
the previous location is printed.

```
main = do
     forkMT ex

ex :: M IO ()
ex  = do name <- lift getHostName
         moveTo "lxtrinder"
         lift $ print name
```

Figure 4: A Simple Strong Mobility Example

The important thing to notice in the example is that, besides the use of `lift`,
the use of a continuation is completely hidden in the monad, and the mobile
thread is written in a similar way as a normal Concurrent Haskell thread.

3.4 Example 2: Mobile Tree Search

The advantage of using strong mobility comes when the programmer wants to
control where the continuation of a computation must be executed. As an ex-
ample, taken from [Sekiguchi 1999], consider the *m*Haskell program in Figure 5.
It is a tree search algorithm, the idea is that there is a network of computers
connected as a binary tree, and the algorithm will transverse the tree and exe-
cute an IO action on each `Leaf`, combining the results with an operator. This is
the typical behaviour of a search robot that analyses web pages by following the
links in them.

In the base case, when `mTSearch` finds a `Leaf` it will simply execute the IO
action. In the next case, `findLoc` is used to extract the next location to be visited
from the right (`treer`) and left (`treel`) branches of the tree:

```
findLoc :: Tree HostName -> HostName
findLoc (Leaf host) = host
findLoc (Node host treer treel) = host
```

```
mTSearch :: IO a -> (a -> a -> a) -> Tree HostName -> M IO a
mTSearch  action op (Leaf host) = lift action
mTSearch  action op (Node host treel treer)= do
       moveTo (findLoc treel)
       x <- mTSearch action op treel
       moveTo (findLoc treer)
       y <- mSearch  action op treer
       return (op x y)
```

Figure 5: Tree search using strong mobility

For each branch, it does a recursive call to mTSearch to search for a Leaf.

One could try to write the same recursive program using remote evaluation as in the example of Figure 6. Looking closely to both versions of the program, it is possible to see that they do not have the same pattern of control transfer between the locations visited. Considering the Tree of Figure 7, where there is a node A with two subtrees B and C. The program in Figure 5 would migrate from A to B, and then to C. The program using remote evaluation (Figure 6) migrates from A → B → A → C → A. While the addition in the first program takes place in C, in the second it takes place in A.

```
mTSearch :: IO a -> (a -> a -> a) -> Tree HostName -> IO a
mTSearch action op (Leaf host) = action
mTSearch action op (Node host treel treer)=  do
    x <- reval (mTSearch action op treel) (findLoc treel)
    y <- reval (mTSearch action op treer) (findLoc treer)
    return (op x y)
```

Figure 6: Tree search using weak mobility

To make the remote evaluation program to have the same behaviour as the one using strong mobility, it would need to have an extra argument representing the continuation of the computation, making the code bigger and more difficult to understand [Sekiguchi 1999]. Hence, the advantage of using mobile threads and the moveTo construct, is that the continuation is hidden in the continuation monad, and the program can be written as a normal Concurrent Haskell program.

3.5 Example 3: The mfold Skeleton

Mobility skeletons are polymorphic higher-order functions that encapsulate common patterns of mobile computation [Du Bois et al. 2005b]. A common pattern of mobility is a computation that visits a set of locations performing an action
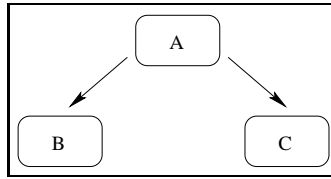
Figure 7: Searching in a Tree

at every location and combining the results. This pattern matches the concept of a distributed information retrieval (DIR) system. A DIR application gathers information matching some specified criteria from information sources dispersed in the network. This kind of application has been considered "the killer application" for mobile languages [Fuggetta et al. 1998]. A skeleton that has this behaviour is shown in Figure 8. The `mfold` skeleton takes as arguments an action (of type `IO a`) to be executed on every host, a function to combine the results of these actions, an initial value, a function that tells what should be done with the result of the computation, and a list of locations to visit.

```
mfold::IO a-> (a-> a-> a)-> a-> (a-> IO ())-> [HostName]-> M IO ()
mfold f op v final hosts = do
 result <-  foldr (liftM2 op) (return v) (map (move f) hosts)
 lift $ final result
  where
  move::IO a -> HostName -> M IO a
  move action host = do
    moveTo host
    result <- lift action
    return result
```

Figure 8: `mfold` using strong mobility

The interesting thing to notice in the implementation of `mfold` presented here, is that although the function `move` is mapped over the list of `hosts` generating a list of actions, the mobility of the thread only occurs when the `foldr` consumes the list, executing the IO actions.

The application of `mfold` to its arguments, generates a mobile thread, hence the program must be executed using `forkMT`, as in the following program:

```
main = do
 mch <- newMChannel
 forkMT (mfold getLoad (+) 0 (writeMChannel mch) listofhosts)
 resp <- readMChannel mch
 print ("Total Load: " ++ show resp)
```

In the example, `mfold` uses `getLoad` to get the load of the hosts in `listofhosts`,

and after visiting all the elements of the list, the result of the computation is sent back through the MChannel `mch`.

4 Case Study: A Mobile-Agent Platform

A mobile agent is a program that can move across locations in a network interacting with resources and other agents. An agent should be autonomous enough in order to decide when and where to move, even when the host that launched the agent is not connected to the network anymore. In this section we describe how *m*Haskell can be used to implement a simple mobile agent platform, based on the Agent Tcl platform [Gray et al. 1996], that supports *partially connected* computers i.e., computers that are not always connected to the network, such as laptops and PDAs. Mobile Agents is an interesting programming paradigm when partially connected computers are involved: a user can launch an agent to do some work and then disconnect his laptop. When connected to the network again, the user can retrieve the information gathered by the agent.

The objective of the simple mobile agent system presented here is to provide the following functionalities:

– Communication between an agent and its creator, and among agents.

– A way of locating and killing agents that are moving on a network

– An agent should be able to find and use resources available in the locations

– The system must be able to handle partially connected machines and its agents

4.1 The Docking System

The mobile agent platform presented here is based on the idea of a *docking system* (Figure 9). Every mobile computer in the network is associated with a permanently connected computer, or *docking station*, that controls and coordinates the mobile agents created by the mobile computer.

The docking station keeps track of the current state of an agent:

```
data AgentState = Located HostName |
     Moving HostName | Killed
```

Every time an agent wants to migrate from one location in the network to another, the docking station must be asked for permission. The docking station contacts the destination and if it is ready to receive the agent, a permission for migration is given and the state of the agent is updated in the docking station. This process is described in Figure 10.

Figure 9: Mobile Agent Platform

```
agentLoop :: MVar AgentState ->
             MChannel DockMesg -> IO ()
agentLoop statea mch = do
 msg <- readMChannel mch
 case msg of
  Move newhost (returnch) -> do
   s <- takeMVar statea
   case s of
    Killed -> do
     writeMChannel returnch Die
     collectGarbage
     return ()
    Located h  -> do
     procMigration returnch newhost statea
     agentLoop statea mch
  (...)
```
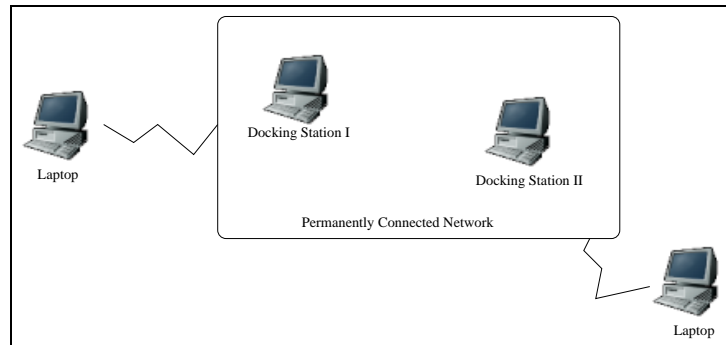
Figure 10: Processing messages sent by the agent

The docking station has one thread to manage each agent registered (`agentLoop`). When the agent executes the `moveTo` primitive, `moveTo` asks the docking station for permission by sending a `Move` message. If the agent was killed by another process, the docking station sends a `Die` message back to the agent, deletes from its internal tables any reference do the agent (`collecGarbage`), and stops the `agentLoop` thread. The current state of the agent is kept inside of an MVar [Peyton Jones 2001], that is a shared mutable variable, and works as a semaphore: every thread that tries to read from an empty MVar will block until it is filled with a value. If the agent is located somewhere, `procMigration` is called (Figure 11).

The `procMigration` function, checks if the remote location is able to receive another agent. If the destination is `Available`, the docking station sends a permission to move and updates the state of the agent. Once the agent arrives, the

```
procMigration returnch newhost statea = do
 resp <- checkRemoteLocation newhost
 case resp of
   Available -> do
      writeMChannel returnch OK
      putMVar statea (Moving newhost)
   NotAvailable -> do
      writeMChannel returnch MoveToDock
      myname <- getHostName
      putMVar statea (Located myname)
```

Figure 11: The `procMigration` function

agent system sends a message back to the docking station confirming its new
location. If the agent needs to migrate to a location that is `NotAvailable`, e.g.
a laptop that is not currently connected to the network, the agent is told to
move to the docking station, and wait until the laptop is connected again to the
network. A location in the network can be `NotAvailable` for a long time and, in
that case, it would be a waste of resources to keep the agent in memory, so its
state, the continuation of its thread, could be saved on disk, using *m*Haskell's
serialisation primitives [Du Bois et al. 2005a], and recovered once the docking
station detects that the location to where the agent wants to move is available.

All locations in the system must run an *agent server* that keeps track of the
agents currently running on that location, and is used, together with the docking
station, to send messages to the agents, as described in Section 4.4.

The `forkMT` and `moveTo` functions have to be modified in order to register the
agent in the docking station once it is created, and to contact it every time the
agent needs to change its current location:

```
type MAgentID = MChannel DockMesg

forkMT :: M IO () -> HostName -> IO MAgentID
forkMT action dockingstation = do
  mch <-  registerAgent dockingstation
  tid <- forkIO (run action)
  registerWithAgentServer tid mch
  return mch
```

The new `forkMT` function registers the agent with the docking station using
`registerAgent`, that will contact the docking station, create a new `agentLoop`
thread for the agent, and return an MChannel that can be used to contact
the `agentLoop` thread. The new agent is created using `forkIO` and its thread id
is registered with the local agent server. The reason for that is explained in
Section 4.4.

An `MAgentID` is simply an MChannel through which it is possible to contact
the `agentLoop` thread for the agent in the docking station. When an agent mi-
grates from the machine that created it, it can only be reached through the

docking station using its `MAgentID`.

4.2 Using Resources

As described in Section 2.1, *m*Haskell already provides primitives for resource registration and discovery, and the same primitives can also be used by an agent to find and use the resources available in the locations that it visits.

We do not describe here how agents find the names for resources and and where they are located. Agents could find these names in distributed databases, or *yellow pages* [Gray et al. 1996], where resources can be registered and accessed as in a peer-to-peer network.
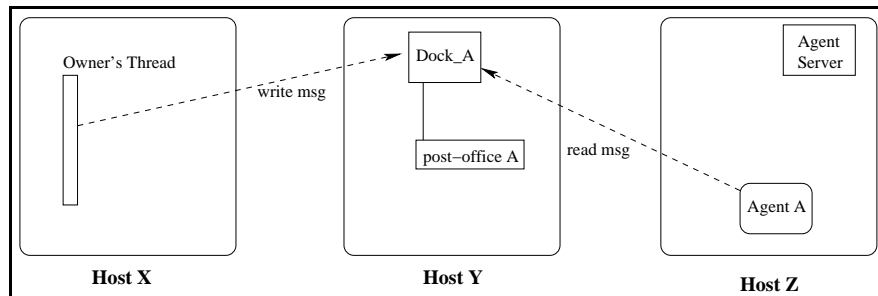
4.3 Agent Communication

Figure 12: The Post-Office

MChannels are single-reader channels, meaning that an agent can only read values from the channel in the location where the channel was created. If the agent migrates to another location, even if it has a reference to a MChannel, it cannot read values from it anymore. Multiple-reader channels for agents (`AgChannels`) can be implemented in *m*Haskell using a centralised server, or, post-office: an MChannel is created in a remote location, the location where the post-office is, and threads reading and writing to the `AgChannel` send messages to this location. The post-office is represented as a thread that reads/writes values into the MChannel, and the primitives for writing and reading values into the `AgChannel` simply send messages to the post-office requesting the operations, as can be seen in Figure 12.

4.4 Locating and Killing Agents

An agent can be easily located and killed through the docking station. For example, here is a function that finds where the current location of an agent is:
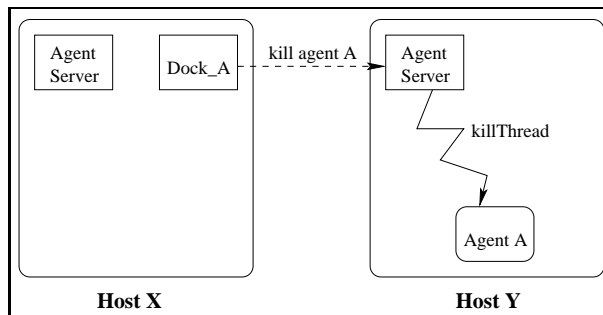
Figure 13: Killing a Mobile Agent

```
pingAgent :: MAgentID -> IO HostName
pingAgent mch = do
  resp <- newMChannel
  writeMChannel mch (Ping resp)
  currentLocation <- readMChannel resp
  return currentLocation
```

It uses the `MAgentID` MChannel to contact the docking station, and ask what the current placement of an agent is, returning an empty name if the agent is not alive anymore. This function is useful when an agent is sent off to visit a large number of locations sequentially, and its owner wants to know approximately the agent's position.

An agent can be killed using the `killAgent` function:

```
killAgent :: MAgentID -> IO ()
killAgent mch = writeMChannel mch KillAgent
```

This function sends a `KillAgent` message to the `agentLoop` thread. The state of the agent is updated to `Killed`, and it will not be allowed to migrate to new locations anymore. Then the docking station tells the agent server in the agent's current location that the agent should die. The agent server uses the `killThread` function, available in the Concurrent Haskell library, to kill the thread in which the agent is running (Figure 13).

5   Conclusions and Future Work

This paper has shown how strong mobility can be elegantly implemented in a language like *m*Haskell with weak mobility, higher-order channels and first-class continuations. Strong mobility has been illustrated using examples and a non-trivial case study. It is well-known in principle that strong mobility can be implemented using weak mobility constructs and continuations [Cejtin et al. 1995, Sumii 2000]. However, we also demonstrate in a realistic case study the usability of the resulting programming style in a modern

functional language such as *m*Haskell, exploiting in particular Haskell's support for monadic programming and higher order functions.

The work could be developed further in several ways. Programs using strong mobility are cumbersome if most of the actions in the mobile thread are monadic, as IO actions have to be lifted into the continuation monad. It would be very useful to add another stage in the Haskell compiler that automatically changes a program of type `IO ()` into a program of type `M IO ()`. Similarly, Concurrent Haskell supports asynchronous exception [Peyton Jones 2001], and the same approach used to kill an agent can be used to raise an exception in a remote thread. The mobile agent system presented here can serve as a model for implementing a library for distributed asynchronous exceptions in Haskell.

Acknowledgements

References

[Bettini and Nicola. 2001] Bettini, L. and Nicola., R. D. (2001). Translating strong mobility into weak mobility. In *Proc. of 5th IEEE Int. Conf. on Mobile Agents (MA)*. Springer-Verlag, LNCS 2240.

[Cardelli 1999] Cardelli, L. (1999). Mobility and Security. In *Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, pages 3–37, Marktoberdorf, Germany.

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

[Claessen 1999] Claessen, K. (1999). A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323.

[Conchon and Fessant 1999] Conchon, S. and Fessant, F. L. (1999). 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.

[Du Bois et al. 2002] Du Bois, A. R., Loidl, H.-W., and Trinder, P. (2002). Thread migration in a parallel graph reducer. In *IFL*. Springer-Verlag, LNCS 2670.

[Du Bois et al. 2005a] Du Bois, A. R., Trinder, P., and Loidl, H.-W. (2005a). *m*Haskell: mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254.

[Du Bois et al. 2005b] Du Bois, A. R., Trinder, P., and Loidl, H.-W. (2005b). Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288.

[Foster and Kesselman 1999] Foster, I. and Kesselman, C., editors (1999). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Friedman and Felleisen 1995] Friedman, D. P. and Felleisen, M. (1995). *The Little Schemer, 4th edition*. MIT Press.

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

[GHC 2006] GHC (2006). The Glasgow Haskell Compiler, http://www.haskell.org/ghc. WWW page.

[Gray et al. 1996] Gray, R. S., Kotz, D., Nog, S., Rus, D., and Cybenko, G. (1996). Mobile agents for mobile computing. Technical Report TR96-285, Dartmouth College.

[Grosso 2001] Grosso, W. (2001). *Java RMI*. O'Reilly.

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

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

[Sekiguchi 1999] Sekiguchi, T. (1999). *A Study on Mobile Language Systems*. PhD thesis, Department of Information Science, The University of Tokio.

[Sumii 2000] Sumii, E. (2000). An implementation of transparent migration on standard scheme. In *Scheme and Functional Programming 2000*, pages 61–64.

[Wadler 1995] Wadler, P. (1995). Monads for functional programming. In J. Jeuring, E. M., editor, *First International Spring School on Advanced Functional Programming Techniques*, LNCS 925, pages 24–52. Springer-Verlag.