Towards Mobility Skeletons

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

[1] School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
{dubois,trinder}@macs.hw.ac.uk
[2] Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany
hwloidl@tcs.ifi.lmu.de

Abstract. In a mobile computation language, programmers have control over the placement of code or active computations across open networks, e.g. programs can migrate between locations. Mobile computations are typically stateful and interact with the state at each location in the network.
We propose *mobility skeletons*, a library of parameterisable functions that encapsulate common patterns of mobile computation. Mobility skeletons are analogous to, but very different from, algorithmic skeletons — parameterisable functions encapsulating common patterns of parallel computation. We have identified three common patterns of mobile computation, and implemented them as a library of higher-order functions in a small extension of Haskell for mobility.
Each such mobility skeleton is defined and illustrated with an example. We show how mobility skeletons can be composed and nested, and illustrate their use in a non-trivial case study: a distributed meeting planner. Mobility skeletons are extensible: there is a small set of mobility primitives, and medium-level abstractions such as remote evaluation can be defined using them. New mobility skeletons can be defined using the medium and low level abstractions.

1 Introduction

Network technology is pervasive and more and more software is executed on multiple locations (or machines) for a variety of reasons. Parallel programs execute parts of a program at different locations to reduce execution time. Distributed programs support interaction at multiple locations, e.g. clients accessing a database server. Mobile programs [7], relocate code or computation in an open network, e.g. a data mining program that visits a series of repositories to extract interesting information. In an open network locations may dynamically join or leave the network.

In addition to specifying a correct and efficient algorithm, parallel, distributed or mobile programs must specify coordination, e.g. how the program is partitioned, how parts of the program are placed in different locations, or how they communicate and synchronise. The coordination can be specified at different levels of abstraction, as illustrated in Table 1. At the lowest level the programmer explicitly controls all aspects of coordination using primitives such as `send` and `receive`. Mid-level abstractions encapsulates several coordination aspects into a single construct, e.g. a Remote Method Invocation (RMI) encapsulates *inter alia* communication from the client to the server, execution of a server method, and communication back to the client. High-level abstractions aim to automatically control most, or even all, aspects of coordination automatically. High level abstractions are highly desirable as they simplify the programmer's task and reuse correct and efficient implementations.

High level abstractions are best developed for parallelism, e.g. implicitly parallel languages like HPF [10], or algorithmic skeletons [5]. Some high level abstractions are now emerging for distributed languages, e.g. behaviours in the Erlang distributed functional language are templates for fault-tolerant distributed programming [1]. This paper describes

Table 1. Abstraction Levels for Distributed Memory Coordination

	Parallelism	Distribution	Mobility
High Level	*skeletons*, HPF	*Behaviours*	*mobility skeletons*
Medium Level	`par`, process networks	RPC, RMI	`reval`, `rfork`
Low Level	`send`, `receive`	`send`, `receive`	`send`, `receive` (MChannels)

mobility skeletons, to the best of the authors' knowledge one of the first high-level abstractions proposed for mobility. Mobility skeletons encapsulate common patterns of mobile coordination as polymorphic higher-order functions.

Mobility skeletons are analogous to, but fundamentally different from, algorithmic skeletons [5] which encapsulate common patterns of parallel coordination as higher-order functions. Most algorithmic skeletons abstract over pure computations in a closed or static set of locations. In contrast mobility skeletons abstract over stateful (or impure) computations in an open network, i.e. a dynamic set of locations. The stateful computations must be carefully managed, in our case using Haskell monads, to preserve the compositional semantics of the mobility skeletons. Moreover, while some mobile coordination patterns are similar to parallel coordination patterns, e.g. an `mmap` broadcasts a computation to be executed on a set of locations, others are different, e.g. `mzipper` repeatedly communicates a computation to prefixes of a sequence of locations.

In Section 2 we review our small extension of Concurrent Haskell for mobility, *m*Haskell. In Section 3, we add another layer of abstraction in *m*Haskell, showing how mobile channels can be used to implement primitives for remote thread creation (`rfork`) and remote evaluation (`reval`). Next, in Section 4, we identify three common patterns of mobile computation and implement them as higher order functions, or *mobility skeletons*. We also demonstrate nesting and composition of mobility skeletons. In Section 5, a case study is presented, where the skeletons are used to implement a distributed meeting planner. Section 6 concludes.

2 Low Level Coordination: mHaskell Primitives

Mobile Haskell [2] (*m*Haskell) is a small extension of the purely functional Haskell language for writing distributed mobile software. *m*Haskell extends Concurrent Haskell [13], a extension supporting concurrent programming, with higher order communication channels called *Mobile Channels* (MChannels). MChannels allow the communication of arbitrary Haskell values including functions, computations (IO actions) and mobile channels.

To preserve the purely functional semantics of Haskell, stateful operations like writing to a file or a channel, are embedded in a monad that encapsulates the state. In particular, stateful or side-effecting computations are embedded in the IO monad and termed *IO actions*. Haskell computations are first-class values: a function can receive IO actions as arguments, return actions as results, and actions can be composed to generate new actions. Hence programs can manipulate computations to generate new abstractions [12], as illustrated by the mobility skeletons in Section 4.

Figure 1 shows the MChannel primitives. 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 and have similar semantics to Concurrent Haskell channels: when a value is written to a channel the current thread blocks until the value is received in the remote host. In the same way, when a `readMChannel` is performed in an empty MChannel, it will block until a value is received on that MChannel. The functions `registerMChannel` and `unregisterMChannel` register/unregister channels in a name

```
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))
```

Fig. 1. Mobile Channels

server. Once registered, a channel can be found by other programs using `lookupMChannel` which retrieves a mobile channel from the name server. The `Maybe` type in Haskell has two values: `Nothing` and `Just a`, so if the lookup finds a MChannel registered with `ChanName`, it returns `Just mchannel`, or returns `Nothing` otherwise. A name server is always running on every *location* (a host in the network that has *m*Haskell runtime system running) 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. Pure functional values are evaluated to normal form before being communicated, but values of type IO are not executed, as explained in [2]. In *m*Haskell, computations are *copied* between machines and no sharing is preserved across machines. An operational semantics for the MChannel primitives is given in [3].

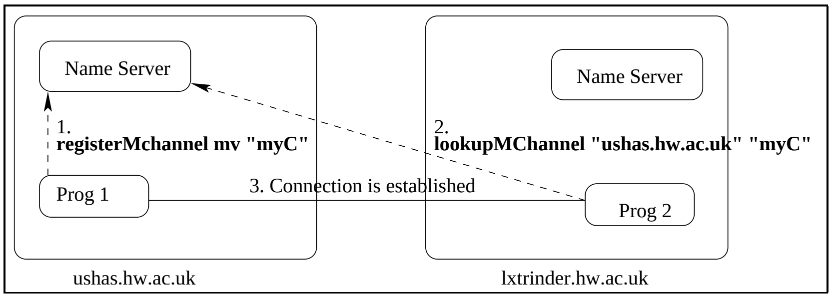

Fig. 2. Example using MChannels

Figure 2 depicts a pair of simple programs using MChannels. First a program running on a location called `ushas` registers a channel `mv` with the name `"myC"` in its local name server. When registered, the channel can be seen by other locations using the `lookupMChannel` primitive. After the lookup, the connection between the two locations is established and communication is performed with the functions `writeMChannel` and `readMChannel`.

2.1 Discovering Resources

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

Figure 3 presents the three *m*Haskell primitives for resource discovery and registration. 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

```
type ResName = String

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

Fig. 3. Primitives for resource discovery

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, 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 type clashes is to use dynamic types. The GHC [8] Haskell compiler has basic support for dynamic types, providing operations for injecting values of arbitrary types into a dynamically typed value, and operations for converting dynamic values into a monomorphic type:

```
toDyn :: Typeable a => a -> Dynamic
fromDyn :: Typeable a => Dynamic -> a -> a
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

The following simple example shows how dynamic types can be used:

```
let list = [ toDyn not,toDyn (id::Int->Int)] in
  let myNot = fromDyn (head list) in
   myNot True
```

In the program, `list` has type `[Dynamic]`. Note that the polymorphic function `id :: a -> a` has to be type casted into a monomorphic type in order to became a dynamic value.

3 Medium Level Coordination: Remote Thread Creation and Remote Evaluation

Programming using MChannels is low level: the programmer has to specify details such as thread creation, communication and synchronisation of computations. In this section we add another layer of abstraction to *m*Haskell by introducing two new functions, one for remote thread creation (`rfork`), and another for remote evaluation of computations (`reval`). These functions have straightforward implementation using MChannels. Their implementation is explained in Appendix A.

A thread can be created in a remote location with the `rfork` function:

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

It takes an IO action as an argument but instead of creating a local thread, it forks a new thread on the remote host `HostName` to execute the action.

In the *remote evaluation* [14] paradigm, a computation is sent from a host A to a remote host B in order to use the resources available in B. It is straightforward to implement *remote evaluation* using mobile channels and `rfork` (Appendix A).

A computation can be sent to be evaluated on a remote location using the `reval` function:

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

The `rfork` function could also be implemented using `reval`. In fact, if we had an implementation of `reval` based on MChannels and not `rfork`, we could say that:

```
rfork comp host = forkIO (reval comp host >> return ())
```

A new local thread is forked (using `forkIO`) so `rfork` will not stay blocked waiting for the result of `reval`. The (`>>`) operator is the monadic operation for combining IO values [12].

The abstraction provided by `reval` is similar to that provided by Java^{TM}'s RMI [9], 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) in order to give access to remote methods.

4 High Level Coordination: Mobility Skeletons

This section identifies three common patterns of mobile computation and implement them as higher order functions, or *Mobility Skeletons* in *m*Haskell, using `reval` and `rfork` from Section 3. While the first mobility skeleton (`mmap`) is analogous to an algorithmic skeleton, even though `mmap` does not evaluate values in parallel, the second and third have no such correspondence.

4.1 `mmap`: Broadcast

A common pattern of mobile computation is to broadcast a computation to be executed on a set of locations. The `mmap` skeleton (see Figure 4) broadcasts its first argument to be executed on every host that is an element of its second argument. It returns a list with the values returned from the remote executions.

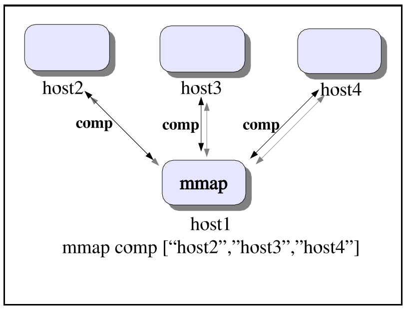

Fig. 4. The behaviour of `mmap`

```
mmap :: IO b -> [HostName] -> IO [b]
mmap f list = do
          result <- mapM (reval f) list
          return result
```

Fig. 5. The definition of the `mmap` skeleton

The implementation of `mmap` (see Figure 5) executes a remote evaluation of its argument `f` on every host of the `list`. The `do` notation in the example is a special syntax for monadic computations [12]. It hides the use of the monadic operations for composing IO actions.

As an example using `mmap` we present a program that gets the load of all locations in a network. If all locations have a `getLoad` function, which returns the load of the location, registered as a resource (`"getLoad"`):

```
registerRes getLoad "getLoad"
```

then the `getLocalLoad` function can be implemented as follows:

```
getLocalLoad :: IO Int
getLocalLoad = do
        res <- lookupRes "getLoad"
        case res of
           Just getLoad -> do
                            load <- getLoad
                            return load
```

This function, when called on a location, looks for a resource named `"getLoad"` and executes it, returning the load of the current location.

Using `getLocalLoad`, we now can write a program that returns a list with the load of all the locations in a network:

```
networkLoad :: [HostNames] -> IO [Int]
networkLoad hosts = do
                     res <- mmap getLocalLoad hosts
                     return res
```

4.2 `mfold`: Distributed Information Retrieval

A common pattern of mobility is a computation that visits a set of locations performing an action at every location and combining the results (see Figure 7). 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 [7].

A *m*Haskell skeleton with this behaviour could have the following type:

```
mfold :: MChannel a -> IO a -> (a -> a -> a) -> a -> [HostName] -> IO ()
```

It takes as arguments an MChannel used to return the result of the whole computation, a action (of type `IO a`) to be executed on every host, a function to combine the result of these actions, an initial value, and a list of locations to visit. Notice that the type resembles the classic function `fold` present in every functional language, that combines the elements of a list using an operator; hence the name of the skeleton is `mfold`.

```
mfold :: MChannel a -> IO a -> (a -> a -> a) -> a -> [HostName] -> IO ()
mfold mch action op value []      = writeMChannel mch value
mfold mch action op value (x:xs)  =
                          rfork (code mch action op value xs) x
      where code mch action op value list =
           do
            value2 <- action
            mfold mch action op (op value2 value) list
```

Fig. 6. The definition of the `mfold` skeleton

The definition of mfold can be seen in Figure 6. In the first case, if the list of locations to be visited is empty, then it simply returns through the MChannel the accumulating argument. If the list of locations to visit is not empty the computation code is run on the head of the list. The function code executes the action on the current host and then does a recursive call to mfold, combining the current value with the result from the execution of action.

As a simple example, mfold can be used to construct an application that computes the total load of a network. Using the getLocalLoad function defined in the previous section, mgetLoad can be implemented as follows:

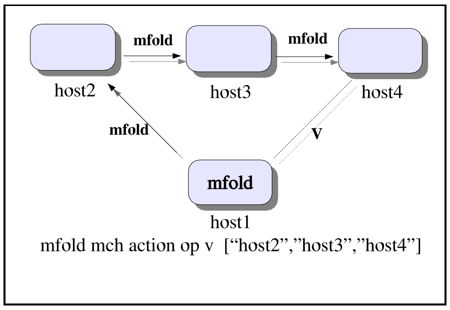

Fig. 7. The behaviour of mfold

```
mgetLoad :: [HostName] -> IO Int
mgetLoad hosts = do
     mch <- newMChannel
     mfold mch getLocalLoad (+) 0 hosts
     load <- readMChannel mch
     return load
```

The mobile function mgetLoad executes getLocalLoad on every location of hosts and combines the results produced on every host using the (+) operator.

We can modify the behaviour of the program just by modifying the arguments passed to mfold. For example, this program:

```
mfold mch getLocalLoad (++) [] listoflocations
```

will collect the load of all the locations in a list, so that the load of the network can be computed later.

Although some of the programs written using mmap can be expressed using mfold, both skeletons have completely different operational behaviours. mfold always executes its continuation on the next location to be visited, while in mmap, the flow of control stays on the location that made the call to it (as can be seen in Figures 4 and 7).

The mfold skeleton is very different than a parallel fold, while in the first the list is used only to indicate to where the computation should move next, in the latter the work is done by splitting its work list into a number of sublists that are then broadcasted to the processors available, and the results of the fold are combined using a parallel divide and conquer algorithm.

4.3 mzipper: Iteration

Another pattern of mobile computation is a computation that visits a sequence of locations, looking for some value that all the locations have to agree with. The value is tested against a predicate on every location, and if it fails, the computation restarts, visiting the sequence of locations from the beginning with a new value, as depicted in Figure 8.

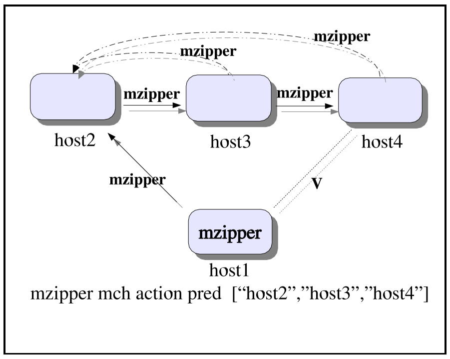

Fig. 8. The behaviour of `mzipper`

As the computation has to move back and forth on the list of locations, the skeleton is called `mzipper` (mobile `zipper`), an analogy to the function `zipper` [11], that describes how to navigate on different data structures.

The `mzipper` template has the following type:

```
mzipper :: MChannel (Maybe a) -> ([a] -> IO ([a],Maybe a)) ->
                    (a -> IO Bool) -> [HostName] -> IO ()
```

It takes a MChannel, that is used to return the value that all the locations agreed (if there is one) back to the place that called `mzipper`, a function that takes a list of values that the locations disagreed in the past and generates a new value, a predicate that indicates if the current location agrees with the current value, and a list of locations to visit. The implementation of `mzipper` is more complex than the one of the other skeletons and is described in Appendix B.

For example, the `mzipper` skeleton can be used to implement a program that keeps visiting locations on a network, and only returns when the load on all the locations that it visited is below a certain threshold. If one of the locations has load above the threshold, it starts visiting the locations again:

```
isLoadBelow :: Int -> [HostName] -> IO Bool
isLoadBelow threshold hosts = do
   mch <- newMChannel
   rBelow mch (myThreshold threshold) isBelowTh hosts
  where
   rBelow mch myTh isBelow hosts = do
     mzipper mch myTh isBelow hosts
     res <- readMChannel mch
     case res of
       Nothing -> rBelow mch myTh isBelow hosts
       Just x  -> return True
   isBelowTh th = (...)
   myThreshold th = (...)
```

The `isLoadBelow` function receives as an argument a `threshold` and a list of locations to visit, and returns `True` when the load on all the locations is below the threshold. The recursive function `rBelow` uses `mzipper` to check if all the locations in the network have the appropriate load. Every time `mzipper` returns `Nothing` through the MChannel `mch`, `rBelow` restarts the search by calling itself. If `mzipper` returns a value, it means that all the locations in the network, at a certain point, had the load under the threshold, and it can return `True`.

The work that is performed by `mzipper` on every location that it visits, is specified by the two locally defined functions `isBelowTh`, that is the predicate, and `myThreshold`, that given a threshold, and the old values of the search, returns a tuple with the old values and the new one.

```
isBelowTh :: Int -> IO Bool
isBelowTh th = do
          load <- getLocalLoad
          return (load<th)
myThreshold :: Int -> [Int] -> IO ([Int], Maybe Int)
myThreshold th list = do
          load <- getLocalLoad
          if (load < th) then (return ([], Just th))
           else (return ([],Nothing))
```

`isBelowTh` uses the previously defined function `getLocalLoad` to get the load of the current location and compares it with the threshold. The `myThreshold` function is used to restart the computation. Given a threshold and a list of old values, it will always return the same threshold if the load of the current location is below the threshold, and returns an empty list as the list of old values is never used in this computation. As `myThreshold` is only called in the first location to be visited, in order to start the computation again, it will return `Nothing` if the load in the current location is not below the threshold. That happens because if the first location in the list can't start the computation, there is nothing else it can do. That is why the `rBelow` function has to call itself again every time `mzipper` returns `Nothing`.

4.4 Nesting and Composing Skeletons

One of the advantages of using a functional language to implement the mobility skeletons, is that it makes easier to compose and nest skeletons in order to model new behaviours. In this section we present examples that explain these concepts.

As an example of nesting, suppose that we have a list of locations that are gateways to networks, and we want to compute the total load on those networks. First, we could implement an IO action that asks the gateways for the hosts in their local network, and then uses `mgetLoad`, which uses `mfold` in its implementation, to compute the load:

```
getMeanLoad :: IO Int
getMeanLoad = do
   res <- lookupRes "myLocations"
   case res of
     Just getMyLocations -> do
             l<- getMyLocations
             r<- mgetLoad l
             return r
```

Then, to compute the loads of the gateways, the programmer just has to broadcast `getMeanLoad` to the gateways:

```
result <- mmap getMeanLoad gateways
```

Stateful computations in Haskell (e.g. mobile computations and skeletons) are always embedded in the IO monad, as discussed in Section 2. IO values are composed using the (>>=) operator. For example, if the programmer wants to calculate the load of a network and then broadcast this value to all the locations, she could compose the mgetLoad function with mmap:

```
getLoadAndBC locations = mgetLoad locations >>=
   λload. mmap (update load) locations
```

or using the do notation [12]:

```
getLoadAndBC locations = do
    load <- mgetLoad locations
    mmap (update load) locations
```

where update updates a resource in all the locations with the load of the network. In fact, all the examples in this text in which the do notation is used, are compositions of IO values.

5 Case Study: The Distributed Meeting Planner

To demonstrate the usefulness of the abstractions presented in the previous sections, we show the development of a larger mobile application. The objective is to demonstrate that mobile applications can be easily constructed using our mobility skeletons, and that almost all the communication is encoded in the skeletons, simplifying mobile programming. The application is a distributed meeting planner. When a user wants to arrange a meeting with several others, she sends a mobile computation that visits the locations of the people involved, trying to find a suitable time.

In the next section, we present a version of the program using mzipper. In Section 5.2, some of the problems of the meeting planner are discussed and a new version using mfold is described.

5.1 A Version Using mzipper

The core of the application is a function called timeMeeting, that takes as an argument a list of locations to visit and returns the time (time here is represented as a String), that everyone agreed for the meeting, if there exists one.

```
timeMeeting :: [HostName] -> IO (Maybe String)
timeMeeting list = do
            mch <- newMChannel
            mzipper mch getNewTime timeOK list
            time <- readMChannel mch
            return time
```

The timeMeeting function, uses mzipper to visit the locations and check for the right time for the meeting. The idea is that, mzipper will look for an empty slot in the time table of the current location, the location that made the call to timeMeeting, and then visit the other locations in the list to check if they agree with the time. If one of the locations doesn't, then it will come back to the first location and ask for a different time. mzipper will return once all the locations agreed with a time, or when the first location doesn't have any other time to suggest.

mzipper is called with three arguments: a MChannel that is used to return the result of the computation back, the getNewTime function that is used every time mzipper needs to find a new time, and timeOK that is executed on every location to check if the location agrees with the current time.

The getNewTime function receives as an argument the list of old times, and returns a tuple containing the same list and a new time if there is one available:

```
getNewTime :: [String] -> IO ([String], Maybe String)
getNewTime oldtimes = do
   ft <-lookupRes "newfreetime"
   case ft of
    Just dyn -> case (fromDynamic dyn) of
                   Just getFreeTimeIO -> do
                      res <- getFreeTimeIO oldtimes
                      return (oldtimes,res)
```

getNewTime looks for a resource (the getFreeTimeIO function) registered in the resource server with the name "newfreetime". This function exists on every location that is running the meeting planner. It checks the local tables of the program to see if there is a new time different from the ones that are in the oldtimes list. Here it is possible to see that mzipper is used in a different way than in the isLoadBelow example. As the computation now uses its old values to compute the new ones, getNewTime always returns the old times in the tuple, while myThreshold just returns an empty list.

As every location has its local copy of getFreeTimeIO (every location has a different time table), it is considered a resource and it must be registered in the resource server on every location so the mobile computation can find it:

```
main = do
      registerRes (toDyn getFreeTimesIO) "freetimes"
      registerRes (toDyn getFreeTimeIO) "newfreetime"
      startUserInterface
```

When the meeting planner is started on every location, the first action that it takes is to register its local resources. As in this case we have more than one resource with different types registered (getFreeTimesIO is used by timeOK), we need to register them as dynamic values using the toDyn function.

The third argument given to mzipper is timeOK, which is executed on every location to check if the current time is suitable.

```
timeOK :: String -> IO Bool
timeOK time = do
  ft <-lookupRes "freetimes"
   case ft of
    Just dyn -> case (fromDynamic dyn) of
                  Just getFreeTimesIO -> do
                        l <- getFreeTimesIO
                        let res = not (isFree time l)
                        return (res)
```

It just looks for a resource called "freetimes", and executes it to get the free times of the current location. Then, it checks (using the isFree function) if the current time for the meeting is included in that list.

Based on the result of this function, the mobile computation decides if it has to migrate to the next host or to go back to the first one to ask for a new time.

Finally, once the application has the time for the meeting, it can broadcast the time to all the locations using `mmap`:

```
mmap (updateTime time) listofhosts
```

where `updateTime` is a function that updates the tables on all the hosts with the time for the meeting.

5.2 Using `mfold`

In the previous implementation of the meeting planner, whenever one of the locations does not agree with the time for the meeting, the computation returns to the first location and restarts the entire search. As every location already has a function that returns all the free times available (`getFreeTimesIO`), the program could be optimised to carry not only one free time, but a list with all the free times available from the first location.

A function like

```
combineStrings :: [String] -> [String] -> [String]
```

that given two lists of strings, computes the intersection of these two lists, could be used to combine the result produced by executing `getFreeTimesIO` on all the locations that will attend to the meeting.

With this optimisation in mind, one could write a new definition for `createMeeting`:

```
createMeeting :: [HostName] -> IO [String]
createMeeting list = do
               mch <- newMChannel
               myfreetimes <- getFreeTimesIO
               mfold mch getLocalTimes combineStrings myfreetimes list
               times <- readMChannel mch
               return times
```

Now, `createMeeting` is described in terms of a `mfold`. It will visit all the locations in `list`, executing `getLocalTimes` on them, and combining the results produced on every host using `combineStrings`.

The `getLocalTimes` function looks for a resource called `"freetimes"`, and returns the result produced by its execution.

```
getLocalTimes = do
  ft <-lookupRes "freetimes"
  case ft of
  Just getFreeTimesIO -> do
      list <- getFreeTimesIO
      return list
```

Finally, as in the previous example, `mmap` can be used to broadcast the time of the meeting to all the participants.

The `mfold` version of the meeting planner is more realistic because the time table that the mobile program carries is small. If the time table is too large to be communicated, e.g. a database, the `mzipper` version would be more appropriate.

6 Conclusions

We have proposed encapsulating common patterns of mobile code as higher-order functions or *mobility skeletons*. While Mobility skeletons are inspired by algorithmic skeletons, which encapsulate patterns of parallel computation on a static set of locations, they differ in several important aspects: mobility skeletons encapsulate the coordination of stateful computations, often describe different coordination patterns from algorithmic skeletons, and are defined on open networks.

Mobility skeletons are implemented as a library of higher-order functions in a small superset of Concurrent Haskell. Mobility skeletons combine stateful computations using monads, allowing the programmer to use familiar notation for the code executed on each machine, retaining the semantics of the underlying purely-functional language, and helping to reason about programs. An operational semantics for mHaskell, based on monadic actions, was proposed in [3].

Features of functional languages, such as higher order functions and polymorphic type systems, make it easier for programmers to abstract over similar patterns of computation, and although many mobile languages are based on the functional paradigm, (e.g. [6, 15, 4]), as far as we know, no one tried to specify common communication behaviours in mobile programming as higher order functions, or skeletons.

Mobility skeletons are easily parameterised, composed, nested and extended using standard monadic composition. The set of primitives for mobility is deliberately kept small, and we have demonstrated how to build higher levels of abstraction, such as remote thread creation and remote evaluation, on top of these primitives. Advanced features of Haskell (present in the GHC compiler), such as as Dynamic Types, and concurrency have been heavily used. As a case study, we presented a distributed meeting planner, which demonstrates the usability of mobility skeletons and the ability to hide the coordination structure in mobile code.

As future work, we plan to investigate cost models for our mobility skeletons to predict when and where computations should migrate to.

Acknowledgements

This work has been partially supported by an UK ORS and Heriot-Watt University James Watt Scholarship, by an ARC grant of the British Council (ARC 1223) and the DAAD (D/03/20257), and by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

References

1. Joe Armstrong. *Making reliable distributed systems in the presence of errors.* PhD thesis, Royal Institute of Technology, Stockholm, 2003.
2. André R. Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Implementing Mobile Haskell. In *Trends in Functional Programming*, volume 4. Intellect, 2004.
3. André R. Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. mHaskell: Mobile computation in a purely functional language. In *Haskell Workshop 04, submitted*, 2004.
4. Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5):704–739, 1995.
5. Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, 1989.

6. Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *ASA'99/MA'99*, Palm Springs, CA, USA, 1999.

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

8. The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, WWW page, January 1998.

9. William Grosso. *Java RMI.* O'Reilly, 2001.

10. High Performance Fortran. http://www.crpc.rice.edu/HPFF/, WWW page, May 2004.

11. Gerard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

12. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.

13. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *The* 23rd *ACM SIGPLAN-SIGACT POPL'96*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

14. Dennis Volpano. Provably secure programming languages for remote evaluation. *ACM Computing Surveys*, 28(4es):176–176, 1996.

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

Appendix A: Definition of `rfork` and `reval`

The `rfork` function can be implemented using MChannels. Suppose that every location in the system also runs a remote fork server (Figure 9). The `startRFork` server creates a channel with the name of

```
startRFork :: IO ()
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 ->do
         writeMChannel nmc io
     Nothing  -> error "rfork: There
                  is no remote server
                  running"
```

Fig. 9. The remote fork server and primitive

the location in which it is running. After that, it keeps reading values from the channel and forking local threads with the IO actions received. If all locations in the system are running this server, we can implement the `rfork` primitive as in Figure 9. The `rfork` function looks for the channel registered in the `startRFork` server, and sends the computation to be evaluated on the remote location `host`.

The `reval` function (Figure 10), uses `rfork` to execute `execMobile` on the remote location. `execMobile` takes two arguments, the first is the actual computation to be executed on the remote host, and the second is a MChannel used to return the result of the computation back. The call to `reval` blocks reading the value from the MChannel until it is sent by `execMobile`.

Appendix B: Definition of `mzipper`

The `mzipper` skeleton (Figure 11) starts by checking if the first location to be visited is the current location. In that case, it asks for the first value to be agreed, using its argument `action`. As the search is just starting, the list of values on which the locations disagreed is empty. The `action` function should return a tuple with first element being the same list that it received as an argument if it is needed for the computation, or an empty list otherwise, and the second is the value that must be

```
reval :: IO a -> HostName -> IO a
reval job host = do
        mch <- newMChannel
        rfork (execMobile job mch) host
        result <- readMChannel mch
        return result
        where
         execMobile :: IO a -> MChannel a -> IO ()
         execMobile job mch = do
           resp <- job
           writeMChannel mch resp
```

Fig. 10. The implementation of `reval`

agreed. If a value is not found, it returns `Nothing` through the channel, otherwise the recursive zipper function is called (`rmzipper`), it takes two extra arguments, the list and the value. If the current location is not the first element of the list then we start `mzipper` again, with the same arguments, on the head of the list.

```
mzipper :: MChannel (Maybe a) -> ([a] -> IO ([a],Maybe a)) ->
                    (a -> IO Bool) -> [HostName] -> IO ()
mzipper mch action pred (fst:hosts) = do
   host <- fullHostName
   if (host == fst)
    then (do
     (l,mv) <- action []
     case mv of
      Nothing -> writeMChannel mch Nothing
      Just v  -> rmzipper v l mch action pred [fst] hosts)
    else (rfork (mzipper mch action pred (fst:hosts)) fst)
 where
 rmzipper :: a -> [a] -> MChannel (Maybe a)  -> ([a] -> IO ([a],Maybe a)) ->
             (a-> IO Bool)  -> [HostName]-> [HostName] -> IO ()
 rmzipper v oldvalues ch action pred oldhosts [] =  writeMChannel ch (Just v)
 rmzipper old oldvalues ch action pred [] (host:hosts) = do
          (l,mv) <- action oldvalues
          case mv of
           Nothing ->  writeMChannel ch Nothing
           Just v  ->  rmzipper v l ch action pred [host] hosts
 rmzipper v oldvalues ch action pred oldhosts (x:xs) =
             rfork (code v oldvalues ch action pred (x:oldhosts) xs) x
 code v oldvalues ch action pred oldhosts hosts= do
      bool <- pred v
      case bool of
        True -> rmzipper v oldvalues ch action pred oldhosts hosts
        False -> do
          let newhosts = (reverse oldhosts) ++ hosts
          rfork (rmzipper v (v:oldvalues) ch action pred [] newhosts)
                (head newhosts)
```

Fig. 11. The definition of the `mzipper` skeleton

The local function `rmzipper` takes two lists of locations as arguments; the first is the locations that were already visited and the second the ones yet to be visited. The base case of `rmzipper` is when the list of already visited locations is empty, meaning that a value was not found and the search has to start again. As before, it looks for a new value using its argument `action`, and if there is no new value it returns with `Nothing`. Otherwise, it continues the search by calling `rmzipper` again and passing the new value to it as an argument.

The second case of `rmzipper` checks if the current location agrees with the current value by using the predicate argument (`pred`). If it agrees, the search continues to the next location, if it doesn't, the list of hosts to visit is recreated as `newhosts` and the search starts again from the beginning.