Towards a Mobile Haskell

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@informatik.uni-muenchen.de

Abstract. This paper proposes a set of communication primitives for Haskell, to be used in open distributed systems, i.e. systems where multiple executing programs can interact using a predefined protocol. Functions are "first class citizens" in a functional language, hence it would be natural transfer them between programs in a distributed system. However, existing distributed Haskell extensions are limited to closed systems or restrict the set of expressions that can be communicated. The former effectively prohibits large-scale distribution, whereas the latter sacrifices key abstraction constructs. A functional language that allows the communication of functions in an open system can be seen as a *mobile computation* language, hence we call our extension *m*Haskell (Mobile Haskell). We demonstrate how the proposed communication primitives can be used to implement more powerful abstractions, such as remote evaluation, and that common patterns of communication can be encoded as higher order functions or *mobile skeletons*. The design has been validated by constructing a prototype in *Glasgow Distributed Haskell*, and a compiled implementation is under construction.

1 Introduction

The last decade has seen an exponential growth in the use of computer networks, both small and large scale networks, e.g. the Internet. One of the impacts of this growth is that almost all programming languages now have libraries or primitives for distributed programming, including functional languages as Erlang [7], Clean [26], Haskell with Ports [12], Eden [1], and GdH [23].

With a huge number of machines connected world wide, researchers start to think about the possibility of using open global networks as a single computing platform [8, 2], sharing computational power and resources. Applications exploiting these new networks require languages with capabilities for building open, distributed systems. Being a distributed system, constructs for exchanging arbitrary data structures are required. Being an open system, the language must provide ways for programs to connect and communicate with other programs and to discover new resources in the network [25].

In a functional language functions are “first class citizens”; e.g. they can be an element of a data structure, be passed as an argument to a function, or returned as the result. Thus it would be natural to expect a communication library for a functional language to allow the programmer to send functions through a network connection for example. A functional language that allows the communication of functions in an open system can be seen as a *mobile computation* language [9, 3], i.e. one that can control the placement of code and active computations across the network [16]. In particular, a mobile program can transport its state and code to another execution environment in the network, where it resumes execution [19]. This paper presents *m*Haskell (Mobile Haskell), a Haskell extension that allows the implementation of distributed mobile programs.

The paper is structured as follows: The next section describes some Haskell extensions and compares them with the concepts of mobile computation. In section 3 we describe the new primitives for distributed communication. In sections 4 and 5 we show that the basic primitives allow the implementation of more powerful abstractions (such as remote evaluation) and that useful patterns of distributed communication can be packaged as higher order functions or *mobile skeletons*, analogous to *algorithmic skeletons* in parallel computing [5]. To validate our ideas and to test the programs presented here, a prototype of our system was implemented using *Glasgow Distributed Haskell* (GdH) and is described in Section 6. In section 7 we discuss some of the issues that must be considered in a real implementation of the system. Finally we present related works, conclusions and future work.

2 Mobility in Haskell Extensions

This section compares some of the Haskell related languages with the concepts of mobile computation presented in the introduction. Mobility in other functional languages is discussed in section 8.

GpH (*Glasgow Parallel Haskell*) [24] is a conservative parallel extension of Haskell, using the parallel combinator `par` to specify parallel evaluation, and `seq` to specify sequential evaluation. Higher-level coordination is provided using *evaluation strategies*, which are higher-order polymorphic functions that use the `par` and `seq` combinators to specify the degree of evaluation of expressions.

Eden [1] is a parallel extension of Haskell that allows the definition and creation of processes. Eden extends Haskell with *process abstractions* and *process instantiations* which are analogous to function abstraction and function application. Process abstractions specify functional process schemes that represents the behaviour of a process in a purely functional way, and process instantiation is the actual creation of a process and its corresponding communication channels [17]. Eden uses a closed system model with location independence. All values are evaluated to normal form before being sent through a port.

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

GDH (*Glasgow Distributed Haskell*) is a distributed functional language that combines features of *Glasgow Parallel Haskell* (GPH) [24] and Concurrent Haskell [15]. GDH allows the creation of impure side effecting I/O threads using the `forkIO` command of Concurrent Haskell but it also has a `rforkIO` command for remote thread creation (of type `rforkIO :: IO () -> PEId -> IO ThreadId`). `rforkIO` receives as one of its arguments the identifier of the PE (processing element) in which the thread must be created. Thus, GDH provides the facility of creating threads across machines in a network and each location is made explicit, hence the programmer can exploit distributed resources. Location awareness is introduced in the language with the primitives `myPEId` which returns the identifier of the machine that is running the current thread and `allPEId` which gives a list of all identifiers of all machines in the program. Threads communicate through MVars that are mutable locations that can be shared by threads in a concurrent/distributed system. Communication occurs by reading/writing values from/to these mutable locations. Any type of value can be read/written from/to MVars, including functions and MVars.

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

Haskell with ports or *Distributed Haskell* [12] adds to concurrent Haskell monadic operations for communication across machines in a distributed system. Thus local threads can communicate using MVars and remote threads communicate using Ports. Ports can be created, used and merged using the following commands:

```
newPort      :: IO (Port a)
writePort    :: Port a -> a -> IO ()
readPort     :: Port a -> IO a
mergePort    :: Port a -> Port b -> IO (Port (Either a b))
```

A Port is created using `newPort`. For a port to became visible to other machines it must be registered in a separate process called *postoffice* using the `registerPort` command. Once registered it can be found by other PEs using the `lookupPort` operation. Ports allow the communication of first order values including ports. All values are evaluated to normal form before sent to a remote thread.

Haskell with ports is a very interesting model to implement distributed programs in Haskell because it was designed to work on open systems. The only drawback is that the current implementation of the language restricts the values that can be sent through a port, using `writePort`, to the basic types and types that can instantiate the `Show` class. Furthermore, the types of the messages which can be received with `readPort` must be an instance of the `Read` class. The reason for these restrictions is that the values of the messages are converted to strings in order to be sent over the network [12].

3 Proposed Communication Primitives

MVars are used for communication between concurrent threads in Concurrent Haskell and for distributed communication in GdH, but their implementation relies on a closed system.

Haskell with Ports on the other hand, extends Concurrent Haskell with monadic operations for communication between machines in a distributed open system. Thus local threads can still communicate using MVars and remote threads communicate using *ports*.

We propose in this text a set of primitives to be used for distributed communication in Haskell. These primitives are similar to those of Haskell with Ports, but without the restriction of communicating just first-order values. We will call our primitives MChannels, *Mobile Channels* (figure 1), to avoid confusion with Concurrent Haskell channels or Distributed Haskell ports.

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

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

Fig. 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 from/to a channel. The functions `registerMChannel` and `unregisterMChannel` register/unregister channels in a name server. Once registered, a channel can be found by other programs using `lookupMChannel` which retrieves a mobile channel from the name server. A name server is always running on every machine of the system and a channel is always registered in the local name server with the `registerMChannel` function.

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

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

Suppose that in the tuple returned by `f x` the first value `a` is a Boolean, the second (`b`) an integer, and the third `c` is a really large data structure (i.e. a big tree). Based on the value of `a` we choose if we want to send the integer `b` (and only `b`) to a remote host. In the example, it seems that the only value being sent is the integer, but because of lazy evaluation that is not what happens. At the point where `writeMChannel` is performed, the value `b` is represented in the heap as the function that selects the second value of a tuple applied to the whole tuple. If `writeMChannel` does not evaluate its argument before communication, the whole value is communicated and is difficult to see that in the Haskell code.

3.1 Sharing Properties

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

If we try to maintain sharing between nodes in our distributed system this will result in a large number of extra-messages and call-backs to the machines involved in the computation (to request structures that were being evaluated somewhere else or to update this structures) that the programmer of the system did not know about. In a typical mobile application, the client will receive some code from a channel and then the machine can be disconnected from the network while the computation is being executed (consider a handheld or a notebook). If we preserve sharing, is difficult to tell when a machine can be disconnected, because even though the computation is not being executed anymore, the result might be needed by some other application that shared the same graph structure.

The problem is also partially solved by making the primitives strict because then expressions will be evaluated only once and only the result of the evaluation is communicated.

3.2 Discovering Resources

In figure 2, we propose three primitives for resource discovery and registration. All machines running mobile Haskell programs must also run a registration service for resources. The scope for the primitives for resource discovery is only the

```
type ResName = String

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

Fig. 2. Primitives for resource discovery

machine where they are called, thus they will only return values registered in the local server. The `registerRes` function takes a name (`ResName`) and a resource (of type `a`) and registers this resource with the name given. `unregisterRes` unregisters a resource associated with a name and `lookupRes` takes a `ResName` and returns a resource registered with that name. To avoid a type clash, if the programmer wants to register resources with different types, she has to define an abstract data type that will hold the different values that can be registered.

A better way to treat these possible type clashes would be to use dynamic types like Clean's *Dynamics* [22], but at the moment there is no complete implementation of it in any of the Haskell compilers.

4 Mobile Examples

4.1 Remote Evaluation

In the *remote evaluation* [27] 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. Suppose that every machine running the system also runs a remote evaluation server that can be implemented like this:

```
remoteEvaluationServer myname = do
                            ch <- newMChannel
                            registerMChannel ch myname
                            loop ch
                             where
                                loop ch = do
                                        io <-readMChannel ch
                                        forkIO io
                                        loop ch
```

The `remoteEvaluation` server takes as argument the name of the current machine and creates a channel with this name. After that, it keeps reading values from the channel and forking threads with the IO actions received. Threads are forked using the `forkIO` function of Concurrent Haskell and the `do` notation in the example is a special syntax for monadic computations [14].

If all machines in the system are running this server, we can implement our remote evaluation primitive as follows:

```
rEval :: IO () -> HostName -> IO ()
rEval io host = do
                  ch <- lookupMChannel host host
                  writeMChannel ch io
```

In fact, remote evaluation usually returns a value as the result of the computation. This is the difference between our `rEval` function and the `revalIO` primitive of GdH. The same behaviour could be easily added to `rEval` by creating another channel to return the result back.

4.2 Simple Code mobility

In Figures 3 and 4 we present a simple example using the new primitives. The program starts on a machine called `ushas` (Figure 4) and "migrates" to a machine called `lxtrinder`, using the `rEval` function. When the thread `mobileCode` is spawned on `lxtrinder`, it obtains the load of the machine using a local function called `getLoad`, that was previously registered in the resource server (Figure 3), and then sends the result back to `ushas` where it is printed.

```
getLoad :: IO Int
getLoad = (...)

main = do
         registerRes getLoad "getLoad"
```

Fig. 3. Code on lxtrinder

4.3 Distributed Information Retrieval

Distributed information retrieval applications gather 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 [9].

The mobile program in Figure 6 visits the machines of a network to calculate the total load of the network (see Figure 5). It is an extension to the previous program: Instead of visiting only one machine in the network it visits a collection of machines.

The program visits all machines in the list `listofmachines` and uses the IO action `getLoad` (of type `IO Int`) locally on all machines. This function must be registered in the name server on all the machines in the `listofmachines`.

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

        recoverError = (...)
```

Fig. 4. Code on ushas

5 Patterns of Mobile Computation

The behaviour of "visiting a collection of machines to perform some action and then return a result", as used in Figure 6, is a very common pattern of mobile computation. We can use the facilities provided by functional languages, in particular higher-order functions, to implement a *skeleton* [5] that abstracts this behaviour.

```
visitSke :: MChannel a -> IO a -> (a -> a -> a) -> a ->
                                                [HostName] -> IO ()
visitSke mch action op load []      = writeMChannel mch load
visitSke mch action op load (x:xs)  =
                          reEval x (code mch action op load xs)
      where code mch' action' op' load' list = do
            load2 <- action'
            visitSke mch' action' op' (op' load2 load') list
```

Using the `visitSke` function we can write the previous program like this:

```
visitSke mch action (+) 0 listofmachines

action = do
          func <- lookupRes  "getLoad"
          case func of
            Just gL -> do
                        load <- gL
                        return load
```

If we want to collect the load of the machines to compute the total load afterwards we just need to change the arguments of the skeleton:

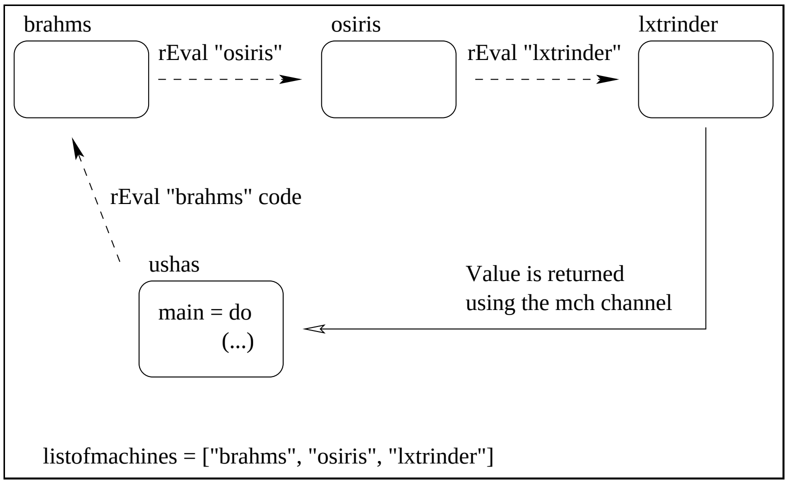

Fig. 5. Distributed Information Retrieval

```
visitSke mch action (++) [] listofmachines
```

6 A Prototype Implementation

To validate the design and to test the examples presented in this text, a prototype of *m*Haskell was implemented using GdH.

In GdH, inter-thread communication is through values that are shared between threads, e.g. an MVar, and there is no other way of establishing a connection between two remote/local threads after their creation.

In our prototype implementation mobile channels are represented as simple MVars and to simulate an open system in GdH we implemented a name server where programs running on different PEs can register their MChannels.

In the example of Figure 7, is possible to see how the GdH implementation of the system works. First an `MChannel` is created on a machine called `ushas` and it is registered with the name `"myC"` in the name server. The name server receives a reference to the MVar created on `ushas` and keeps it in a table together with its name. After the registration process has finished the PE called `osiris` can send a message to the name server asking for the channel that was registered with the name `"myC"` using the `lookupMChannel` command.

How do the primitives find the name server? Threads in GdH can only communicate if they have a reference to a value that is shared between them. We solved the problem by adding an extra parameter to the remote communication primitives that is the channel in which the name server waits for connections. This channel is a *Concurrent Haskell* channel, that is implemented as a linked list of MVars, hence the name server can handle more than one client sending requests at the same time. Programming with this extra parameter can be cum-

```
main = do
      mch <- newMChannel
      mobileCode mch 0 listofmachines
      resp <- readMChannel mch
      print ("Total Load: " ++ show resp)

mobileCode :: MChannel Int -> Int -> [HostName] -> IO ()
mobileCode mch load []      = writeMChannel mch load
mobileCode mch load (x:xs)  = rEval x (code mch load xs)
      where code mch' load' list = do
              func <- lookupRes  "getLoad"
                case func of
                    Just gL       -> do
                                       load2 <- gL
                                       mobileCode mch' (load' + load2) list
```

Fig. 6. Collecting Load Information from several Machines

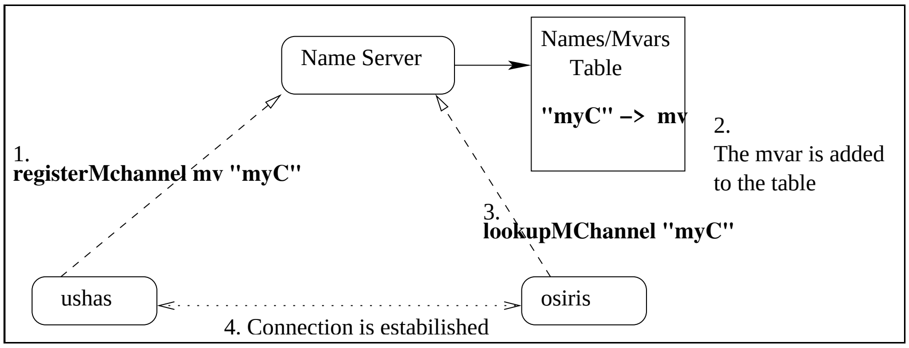

Fig. 7. GDH implementation of Mobile Channels

bersome in some cases but it serves to our purpose of simulating an open system using GDH.

Although we do not need a registration service for resources in GDH, because of the shared heap, we simulate it in the same way as the name server. In fact we only have one name server that accepts registration for both resources and channels but keep them in different tables.

The strict primitives can be partially simulated in our GDH prototype by using *evaluation strategies* [24]:

```
writeMChannel ch v = rnf v `seq` lazyWriteMChannel ch v
```

Here the strategy `rnf` evaluates its argument `v` to normal form before it is written into the channel `ch`. But this only works if the value being communicated was instantiated in the `NFData` class, which is a Haskell class that describes how to evaluate values.

7 Implementation Design

The GdH prototype allows us to test the concepts and ideas of the new communication primitives but is not suitable for the implementation of real applications. In this section we discuss the main issues that must be considered in a real implementation of the mobile system proposed.

Mobile systems have to abstract over the heterogeneity of large scale distributed systems, allowing machines of different architectures running different operating systems to communicate. This abstraction is usually achieved by compiling programs into architecture independent byte-code. As a platform to build our system, we have chosen the *Glasgow Haskell Compiler* (GHC) [10], a state of art implementation of Haskell. The main reason for choosing GHC is that it supports the execution of byte-code combined with machine code. GHC is both an optimising compiler and a interactive environment called GHCi. GHCi is designed for fast compilation and linking, it generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. Both GHC and GHCi share the same runtime-system, based on the STG-machine [13], that is a graph reduction machine.

In our implementation of the mobile system we are planning to implement routines for packing and unpacking Byte-Code Objects (BCOs), that are GHC's internal representation for its architecture independent byte-code, in the same way that other heap objects are communicated in GpH, GdH and Eden. By packing we mean *serialising* these objects into a form that is suitable for communication. As the basic modules in GHC are compiled into machine code and are present in every standard installation of the compiler, the packing routines have to pack only the machine independent part of the program and link it to the local definitions of the machine dependent part when the code is received and unpacked. This gives us the advantage of having much faster code than using only byte-code. Once packed, the BCO can be communicated using a simple TCP/IP socket connection. All machines running the mobile programs should have the same version of the GHC/GHCi system with an implementation of the primitives for mobility and also have the same binary libraries installed. Programs that communicate functions that are not in the standard libraries must be compiled into byte-code using GHCi.

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

In the future it may be possible to extend the compiler with a *mobility analyses* (maybe based on a non-determinism analyses [21]) that would decide the parts of the program that should be compiled into byte-code and the parts that could be compiled into machine code.

8 Related Work

There are other extensions to functional languages that allow the communication of higher-order values. Kali-Scheme [4] and Erlang [7] are examples of strict untyped languages (Erlang is dynamically typed) that allow the communication of functions. Haskell is a statically typed language hence the communication between nodes can be described as a data type and many mistakes can be caught during the compilation of programs. Other strict typed languages such as Nomadic Pict [28], Facile [18] and Jocaml [6] implement the communication primitives as side effects while we integrate them to the IO monad hence preserving referential transparency.

Curry [11] is a functional logic language that provides communication based on Ports in a similar way to the extension presented in this paper. Another language that is closely related to our system is Famke [26]. Famke is an implementation of threads for the lazy functional language Clean [20] (using monads and continuations), together with an extension for distributed communication using ports. The problem of their approach is that they do not have a real implementation of concurrency, their system provides interleaved execution of atomic actions implemented using continuations.

9 Conclusions and Future Work

In this paper we have presented a set of primitives for communication in a distributed extension of Haskell. Our primitives differ from previous approaches in that they allow communication of arbitrary expressions, including functions, in an open system. We have also demonstrated that our primitives, together with Haskell's higher-order functions, allow the implementation of powerful abstractions, e.g. remote evaluation, and useful patterns of communication can be packaged up as higher order functions or *mobile skeletons*. To validate our design and to test the programs presented in this paper, a prototype of our system was implemented using GdH.

We are currently working on a more robust implementation of the system. We will extend the GHC compiler and its byte-code interpreter GHCi with routines for packing and communicating heap objects as described in section 7. The novelty of the system lies in its ability to communicate the byte-code representation of heap objects generated by GHCi, thus allowing programs running on an heterogeneous network to communicate.

References

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

2. Luca Cardelli. Abstractions for mobile computation. In *Secure Internet Programming*, pages 51–94, 1999.
3. Luca Cardelli. Mobility and Security. In *Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, pages 3–37, Marktoberdorf, Germany, August 1999.
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*. MIT Press, 1989.
6. Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.
7. Erlang. http://www.erlang.org/, WWW page, May 2003.
8. Ian Foster, C. Kesselman, and S.Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.
9. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.
10. The Glasgow Haskell Compiler. http://www.haskell.org/ghc, WWW page, January 1998.
11. M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702, pages 376–395. Springer LNCS, 1999.
12. Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with ports. In *Implementation of Functional Languages*, volume 2011, pages 107–121. Springer LNCS, 2000.
13. Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
14. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.
15. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
16. Zeliha Dilsun Kirli. *Mobile Computation with Functions*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2001.
17. Ulrike Klusik, Yolanda Ortega-Mallen, and Ricardo Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages*, volume 1595, pages 103–119. Springer LNCS, 1999.
18. Frederick Colville Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie mellon University, 1995.
19. Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 3(42):88–89, March 1999.
20. Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent Clean. In Leeuwen Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, volume 505, pages 202–219. Springer LNCS, 1991.

21. R. Peña and C. Segura. A polynomial cost non-determinism analysis. In *Implementation of Functional Languages*, volume 2312, pages 121–137. Springer LNCS, 2002.
22. Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185. Springer LNCS, 1998.
23. R. Pointon, P.W. Trinder, and H-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *Implementation of Functional Languages*. Springer LNCS, 2000.
24. Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
25. P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 2002.
26. Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In *Implementation of Functional Languages*, volume 2670. Springer LNCS, 2003.
27. Dennis Volpano. Provably secure programming languages for remote evaluation. *ACM Computing Surveys*, 28(4es):176–176, 1996.
28. Pawel Tomasz Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Wolfson College, University of Cambridge, 2000.