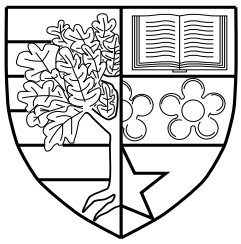
Mobile Computation in a Purely Functional Language

by

André Rauber Du Bois



Submitted for the Degree of
Doctor of Philosophy
at Heriot-Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
May 2006

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

André Rauber Du Bois (Candidate)

Phil Trinder (Supervisor)

Date

Abstract

In *mobile programs*, computations can move, or be moved, over a network to better utilise the available resources. Mobile languages may be *implicit* or *explicit*. In *implicit mobile languages* computations are moved automatically by the runtime system of the language. This kind of mobility is usually exploited in small distributed systems, e.g., parallel systems running on a cluster of machines with equal capabilities. *Explicit mobile languages* give the programmer control over the placement of active computations and execute on open networks where programs can join and leave the distributed system.

This dissertation investigates purely functional *mobile languages*, making the following contributions:

Implementing and evaluating implicit mobility in a purely functional language: Semi-explicit parallel functional languages usually have automatic mechanisms for the distribution of potential work, i.e., unevaluated expressions. In these systems, it is common for some processors to be idle while others have many runnable threads. The performance of these systems can be improved if besides potential work, threads can also be migrated. The GUM runtime system that supports parallel Haskell variants has been extended with a thread migration mechanism. Migrating a thread incurs significant execution cost and the system implemented uses a sophisticated mechanism to choose when to migrate threads. Measurements of nontrivial programs on a high-latency cluster architecture show that thread migration can improve the performance of data-parallel and divide-and-conquer programs with low processor utilisation.

The design, implementation and evaluation of a purely functional language with explicit mobility: The thesis proposes, implements and evaluates *m*Haskell, a Haskell extension for distributed mobile programming. It is the first purely functional explicitly mobile language, and extends Concurrent Haskell with *mobile channels* (MChannels) and a set of low-level primitives. MChannels are higher-order, single-reader communication channels that allow the communication of any Haskell value including functions and channels. MChannels provide *weak mobility*, i.e., the means to start new computations on remote locations. An operational semantics for the *m*Haskell primitives is given. Programming with MChannels is low-level: the programmer has to specify aspects such as communication and synchronisation of computations, and if mobile values

are not carefully managed, e.g., using dynamic types, type errors may occur at runtime. Conventional medium-level abstractions for mobile computation such as *remote thread creation* and *remote evaluation* are readily defined in *m*Haskell, using Haskell's first-class computations. *Strong mobility* is defined as the combination of weak mobility, higher-order channels and first-class continuations.

Proposing, implementing and demonstrating new high-level mobile coordination abstractions, mobility skeletons: Analogous to algorithmic skeletons for parallelism, mobility skeletons are higher-order polymorphic functions encapsulating common patterns of mobile computation. However mobility skeletons are different from algorithmic skeletons as where the latter abstract over pure computations in a closed or static set of locations, mobility skeletons abstract over stateful computations in an open network. The range of *m*Haskell abstractions have been used to implement both conventional mobile applications, such as a distributed meeting planner and a stateless web-server, and sophisticated mobile applications. The latter includes a mobile agent system with location independent communication and distributed asynchronous exceptions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Networks are increasingly pervasive, hence a large number of programming languages have libraries or primitives for distributed programming, including functional languages as Erlang [Erl06], Clean [vWP02], Haskell, [FH01, PTL00], OCaml [CF99], and Scheme [CJK95].

Researchers are investigating the possibility of exploiting the computational power and resources available in global networks [FKT01, Car01]. One way of using the resources available on both local and global networks are *mobile computation* languages [Car99, FPV98, Kir01]. In languages that support mobile computation, executing computations can move in the network in order to better use the resources available in it. Basically a mobile program can transport its state and code to another location in the network, where it resumes execution [LO99]. Mobile computations may be a user application or an autonomous mobile program such as an agent. While a mobile agent can migrate anywhere on the Internet, active applications, processes or threads, typically migrate in a local cluster of computers [MDW99].

Languages that support *explicit* mobile computation enable the programmer to control the placement of active computations and execute on open networks, i.e., a network where locations, or machines, can dynamically join and leave the computation. These mobile computations are typically stateful and interact with the state at each location. In *implicit* mobile programs, running processes are moved automatically by the runtime system of the language, e.g., for load balancing.

1.1 Mobility in Programming Languages

Mobility is about physical and logical computing entities that move [MDW99], i.e., mobile computers and mobile programs. Although both subjects are related, this dissertation focuses on logical mobility, or mobile languages. Mobile languages can be classified based on *which* entity determines the move. Mobile languages may have *implicit* or *explicit* mobility:

- *Implicit*: The main idea of implicit mobility is to *automatically* move an executing computation/process/thread from one location in the network to another, and it is most commonly used in parallel systems to reduce runtime. Implicit mobility is typically exploited in small systems such as a single cluster or LAN [MDW99], but with the increasing popularity of GRID computing [FKT01], it is now being used in larger scale networks, e.g. [ZTML05].

- *Explicit*: Explicit mobile languages give the *programmer* control over the placement of active computations, and execute on open systems. In an explicit mobile language a computation, i.e., state and code of a program, can move from one location to another, where its execution is completed.

To summarise, implicit mobility is exploited in small networks or clusters (closed systems) and explicit mobility is exploited in Internet-scale networks (open systems).

1.2 Mobile Functional Programming

Functional languages aggregate several characteristics that are also important for mobile computation languages, and as a consequence, many mobile languages are based on a functional language, e.g., Facile [Kna95], Jocaml [CF99], Kali [CJK95], Tube [Hal97], etc. The characteristics are as follows:

Higher Order Languages: A language for mobile code must provide abstractions that allow mobile code to be identified and used within the language. For example, a language for mobile agents should provide a primitive that identifies the code that is going to migrate, and a reference to this code can be used in a program to pass it as an argument to functions, to execute it, or to communicate it through a network connection [Kna95]. The higher order nature of functional languages already provides a

way of manipulating code within the language: functions are first class citizens, which means that functions can receive functions as arguments, return functions as the result of a computation and data structures can contain functions as elements. Thus its natural to think that a communication library for a functional language should allow the programmer to send functions through a network connection.

Support for Defining High Level Abstractions: From basic primitives, the user of functional languages can compose and generate new abstractions that model with more clarity the problem being solved [Hug89]. Examples of these abstractions include evaluation strategies for parallel and distributed programming [THLP98, PTL00], Skeletons for parallel programming [Col89] and abstractions for concurrent [PJGF96] and distributed programming [FH01].

Formal Reasoning: Purely functional languages are based on well-understood computational models which helps formal reasoning. Such formal foundations also facilitate formal analyses of the programs (e.g., non-determinism, cost, granularity). Certainly would be interesting to adapt these analyses for mobile computation programs. For example, non-determinism analyses (e.g., [PS02]), aim to identify the source of non-determinism in the program and how it might affect other parts of the program. The same kind of analyses could be modified to identify the sources of mobility in the program. This information could then be used by the compiler to optimise code generation for mobility.

1.3 Purely Functional Languages

In purely functional languages, especially those with non-strict semantics, an expression can be evaluated in any order without affecting the semantics [JH93]. This characteristic was exploited in several extensions for parallel computation in functional languages like Haskell (for a survey on parallel extensions of Haskell the reader should refer to [TLP02]). This also means that, in a language with *implicit* mobility, running threads can be reallocated in order to better use the processors available.

The objective of an *explicitly* mobile program is usually to exploit the resources available at specific locations. Resources include databases, programs, or specific hardware. As a result mobile programs are usually stateful and in a pure language must be carefully managed to preserve referential transparency. This is in contrast to parallel

functional languages, where stateless computations are freely distributed across locations to reduce runtime. In [PJW93], Peyton-Jones and Wadler presented the use of *monads* [Mog89] to specify stateful computations in Haskell. Stateful operations are encapsulated inside of an abstract data type of I/O actions. Functions can receive actions as an argument, return actions as a result, and actions can be *glued* together to generate new actions. Hence in Haskell computations are first-class values, and the purely functional part of the language can still be used to write programs that manipulate computations generating new abstractions in the language.

1.4 Contributions of this Thesis

The objective of this dissertation is to investigate the use of a purely functional language in the development of systems that use mobile computation. The research described in this dissertation has made the following contributions:

- **The Design, Implementation and Evaluation of an implicit mobile system** [DBLT03]: To support high level coordination, parallel functional languages need effective and automatic work distribution mechanisms. Many implementations distribute potential work, i.e., sparks or closures (unevaluated expressions), but there is good evidence that the performance of certain classes of programs can be improved if current work, or threads, are also distributed. Migrating a thread incurs significant execution cost and requires careful scheduling and an elaborate implementation. In Chapter 3, we describe the design, implementation and performance of thread migration in the GUM runtime system [THM+96] underlying Glasgow parallel Haskell (GpH) [THLP98]. Measurements of nontrivial programs on a high-latency cluster architecture show that thread migration can improve the performance of data-parallel and divide-and-conquer programs with low processor utilisation.

- **The Design and Implementation of an Explicitly-Mobile Language** [DBTL03, DBTL04a, DBTL05a]: *Mobile Haskell* (*m*Haskell) is a small extension of the purely functional language Haskell for writing distributed mobile software. *m*Haskell extends Concurrent Haskell [PJGF96], an 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. Stateful or side-effecting computations, such as the communication of values, are embedded in the IO monad, to retain the purely functional nature of the language. As a specification, an operational semantics for MChannels is given, based on previous work on semantics for IO actions [PJ01, MPJMR01].

- **The Design, Implementation and Evaluation of Mobility Skeletons, Higher-Order Abstractions of Common Mobile Coordination Patterns** [DBTL04b, DBTL05c]: Programming using MChannels is low level: the programmer has to specify details such as thread creation, communication and synchronisation of computations and if mobile values are not carefully managed (e.g., using dynamic types), type errors may appear at runtime. To make mobile programming easier we have implemented, using MChannels, mid-level constructs for weak mobility, such as remote thread creation and remote evaluation, and also higher level constructs called *Mobility Skeletons*. Mobility Skeletons are higher-order functions that encapsulate common patterns of mobile computation.

- **Strong Mobility Using Weak Mobility and a Continuation Monad** [DBTL05b]: Monadic programming is a powerful way of describing new abstractions in a functional programming language [Wad95]. Using a continuation monad and MChannels we describe a new way of implementing Strong Mobility (explicit thread migration) in a functional language. As concurrent actions in Haskell are already in a monad, the system described here, can be easily used in combination with Concurrent Haskell.

It is expected that the work presented here may help in the development of new commercial technology for distributed mobile programming.

1.5 What is not covered by this dissertation

This thesis does not cover the following subjects:

Security and Safety: One of the main aims of mobile computation is to share resources among the participants of a distributed system. In order to obtain the

advantages of mobile computation *safety* (received computation should not be the cause of runtime errors that may prevent the program to present its expected behaviour) and *security* (protection against malicious code) must be considered. Enforcing safety and security through extended type systems is an active research area [Kir01, AGH+04, Tho97]. This dissertation focuses mainly on how to express mobility of computations in a purely functional language. It is possible that security and safety could be provided in a lower level of the system, i.e., the runtime system, and not in the programming language constructs. The last Chapter of this text discusses how the system described here could be extend to support some level of security and safety.

Mobile Modules: A key issue in a mobile language is to control how much code moves, e.g., should primitives for addition and printing be copied? If the complete representation of the computation is not communicated, the mobile code must dynamically link to code at the destination location. Mobile Haskell provides a simple means of controlling mobility: only modules compiled to byte code are communicated. Modules like the prelude that are compiled to machine code are not communicated, and dynamically linked. The security and safety issues of dynamically linking mobile code have been much studied, e.g., [Sew01, SLW+04], and although not currently implemented, some of these techniques could be incorporated into *m*Haskell.

1.6 Outline of Dissertation

In Chapter 2 we present the main mobility concepts. In Chapter 3, we describe the design, implementation and evaluation of a system for automatic thread migration in the GUM [THM+96] runtime system that is the implementation of three parallel Haskell extensions, namely Eden [BLOMP97], GpH (*Glasgow Parallel Haskell*) [THLP98] and GdH (Glasgow Distributed Haskell) [PTL00]. We demonstrate that in a high-latency cluster of machines automatic thread migration can reduce the run time of non trivial parallel programs with low processor utilisation. Chapter 4, reviews Monads and how IO and concurrency can be expressed in a purely functional language. Chapter 5, describes *m*Haskell, an extension for explicit mobile programming in the purely functional language Haskell. We describe the basic primitives of the language - called MChannels, discuss the design decisions, and its implementation. Finally we give an operational

semantics for MChannels based on previous work on semantics for IO monadic computations. Chapter 6 shows that MChannels can be used to implement higher level abstractions for mobility, such as remote thread creation and remote evaluation, *Mobility Skeletons* and Strong Mobility. Chapter 7, presents three case studies that demonstrate that the abstractions designed in Chapter 6 can help to implement traditional mobile applications such as a distributed meeting planner, a stateless web server and a mobile agents system. Chapter 8 presents conclusions and future work.

1.7 Publications

Some of the work described in this dissertation has been published:

- André R. Du Bois, Phil Trinder and Hans-Wolfgang Loidl. *m*Haskell: mobile computation in a purely functional language. *Journal of Universal Computer Science*. 11(7):1234-1254, Springer/Knowledge Centre, 2005.
- André R. Du Bois, Phil Trinder and Hans-Wolfgang Loidl. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273-288, 2005.
- André R. Du Bois, Phil Trinder and Hans-Wolfgang Loidl. *Implementing mobile Haskell*, In *Trends in Functional Programming*, vol. 4, pages 79-94. Intellect Books, 2004.
- André R. Du Bois, Phil Trinder and Hans-Wolfgang Loidl. Towards a Mobile Haskell. In *Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, p. 102-116, 2003.
- André R. Du Bois, Hans-Wolfgang Loidl and Phil Trinder. Thread Migration in a Parallel Graph Reducer. In *Implementation of Functional Languages*. Springer-Verlag, LNCS 2670, pages 199-214, 2002.

Chapter 2

Mobile Computation

2.1 Introductory Mobility Concepts

Mobile computations are logical computing entities that can migrate in a network [MDW99]. Examples include a user program that moves between locations in a network in order to use the processing power available, or a mobile agent, i.e., an application that migrates to act on behalf of its owner.

Mobility related terms can have different meanings depending on the research area. The hardware community usually relates the term *mobility* to the mobility of physical devices like laptops, notebooks and palm computers, while the software community relates it to programs or code that can move between locations. To avoid this confusion Luca Cardelli proposed that physical mobility (mobile hardware) should be called *mobile computing* while virtual mobility or mobility of software should be called *mobile computation* [Car99]. This thesis focuses on the latter form of mobility, although there is a strong connection between both areas: mobile computing can benefit from the use of mobile computation, as described later in this Chapter.

Another term that has many different meanings is *mobile agent*. In this dissertation, the term mobile agent is used to mean mobile computation with some autonomy (i.e., it decides when it should move), that when migrates, takes with it its code, data, thread state and resumes execution at a new location.

Mobile computation can be further classified in terms of *which* entity decides what is going to migrate:

Implicit Mobile Computation: Occurs when the system (e.g., operating system

or the runtime system of the language in use) decides to *automatically* migrate running computations.

Explicit Mobile Computation: Occurs when the *programmer* controls the placement of code and active computations in a network. Programming languages that give that kind of control to programmers are called *mobile languages*.

The key benefits of mobile computation are the *ability to move toward a desired resource* and *improved flexibility* [MDW99]:

Moving Toward a Desired Resource: By accessing data or resources locally, performance can be improved in many cases. For example, in a system with implicit mobility, computations can be moved to an lightly loaded computer, or with explicit mobility, a computation could choose to move closer to a database server to avoid the use of remote queries.

Improved Flexibility: Which means easier reconfiguration or improved reliability. For example, a parallel system with automatic parallelisation can correct a poor initial load balancing by migrating tasks to idle processors in the network. In a system with explicit mobility, a computation can be sent to a server in order to increase the services provided by that server.

There are three different implementation models for moving code and computations [Car97]:

- *Moving Text*: the source text of a program is sent over a network connection and interpreted on a remote host. This model can be easily implemented in any interpreted language. The context of the computation in the original location, i.e., network connections, state of the program, is lost and must be restarted at the destination location.

- *Moving Byte-Code*: Byte-code is compiled code generated for an abstract machine. It is faster to execute and communicate than moving text, but it is just another representation for the source code.

- *Moving Closures*: A closure is not only the code for the application but also the *context* in which it was executing in the original location. A closure contains the code and the state of the computation. The state may contain network connections, the stack of the current thread and registers and other things needed to restart a computation on a remote location.

The first two models are usually referred as *mobile code* and the the third as *mobile computation.*

2.2 Early Works on Mobile Computation

The idea of mobility is not really new, several different kinds of mobility have been used for many years, for example, process migration and code mobility. Here we review some of these early work.

2.2.1 Process Migration

In the late 1970s process migration in operating systems was an active research area. *Process Migration* is transferring a running process from one machine to another. A process is an operating system abstraction that encompasses code, data, and operating system state associated with a running application [MDW99]. The objective of introducing process migration in the operating system level was to achieve better load balancing and to support fault tolerance. Process migration is one form of *implicit mobility*, is *transparent* to the user and is not under the control of the programmer. The operating system decides when the process should be migrated. Many operating systems were designed to support process migration, some well known examples are Mach [BRS+85], MOSIX [BL98] and Sprite [OCD+88].

2.2.2 Code Mobility

PostScript[TW85] and SQL[GW99] are two examples of mobile code systems that are widely used. PostScript documents are code descriptions of documents that are sent to printers through a network, where they are then executed to generate the desired output. PostScript documents are usually smaller than the documents that they generate when executed on the printer, thus in this case the use of PostScript saves network traffic. Another advantage is that PostScript documents work like a common language between different printers and applications.

SQL is a database query language. Client programs often connect to database servers and send SQL queries to be evaluated by the server. When dealing with large databases this client/server architecture reduces network usage compared to duplicating in the clients the data held by the servers.

Java Applets are another common mobile code application [Fla99]. Java Applets are byte code programs that are downloaded from web servers together with web documents and are executed in a web browser, giving a dynamic behaviour to web pages. Java Applets are programs written in the Java language [Fla99] and compiled into Java Virtual Machine [LY99] byte code, which must be supported by the web browsers.

2.3 Implicit Mobile Computation

Automatic migration of active computations in a small scale network has the following key benefits [MDW99]:

Load distribution: a heavily loaded computer can offload a subset of its active processes to idle nodes.

Fault tolerance: the system might detect a partial failure in one of the nodes (e.g., memory, disk) and migrate a running program to another node in the network for it to finish execution.

Improved Locality: a computation can be moved closer to the resources it needs to access (e.g., databases, processing power).

Implementing process migration at the operating system level, as described in section 2.2.1, can involve fairly complex modifications to the operating system. Research has addressed the problem by providing migration at a higher level in the system, or at user level, in migration packages such as Condor [LLM88] and Emerald [SJ95]. In both, mobility is implemented entirely as a user-space mechanism. This produces a more maintainable and portable mechanism than working at the operating system level.

2.3.1 Parallel Functional Programming

Purely functional languages have good potential for parallelism. Because of referential transparency, computations can be evaluated in any order without affecting the semantics. The lack of side-effects makes it possible to evaluate subexpressions of a program in parallel without any risk of interference [JH93]. Furthermore, pure languages are easier to reason about, parallel semantics are easily developed, and the programs are amenable to program derivation and optimisations by transformation of the code [TLP02].

Many parallel functional languages have semi-explicit parallelism, where the programmer only indicates what computations could be run in parallel and the compiler or RTS of the language decides when and where these expressions could be evaluated. The problem with this automatic approach for load balancing of tasks is that a bad initial allocation of tasks can ruin the parallelism of the program. However a poor load balance may be corrected by using thread migration. Having referential transparency in the language also means that, in a language with *implicit* mobility, running threads can be reallocated in order to better use the processors available, without affecting the result of the computation.

2.4 Explicit Mobile Computation

The main difference between mobile languages and the early works on simple code mobility and process migration lies in *which* entity determines the migration. Mobile languages give the *programmer* the ability to express when and where computation or code should move, whereas in implicit code mobility or process migration, code and state of the computation are moved by an *external* entity (i.e., a distributed operating system decides to migrate a process because of the high load of one processor or a postscript (code) document is sent to the printer).

In [FPV98], the authors introduce a basic classification for explicit code mobility in which they define the concepts of *Strong Mobility* and *Weak Mobility*.

Weak Mobility is the ability to move only code from one machine to another. With weak mobility two cases can arise: in the first model a program running in a machine links the incoming code dynamically and executes it, and in the second model 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 [CPV97]. In [Car01], Cardelli gives a classification for strong mobility based on the way that the programs treat active network connections:

- *Computation*: is the ability of migrating code together with the context of its

execution. The context may include data, execution state and active network connections, which are kept on transmission.

- *Agents*: An Agent can move to a locations carrying its context, but is self-contained, i.e., they do not communicate remotely, rather they move to some location and communicate locally when they get there.

Many strong mobile systems aim to provide what is called *Transparent Migration* [Sek99]:

Transparent Migration: Migration is called *transparent* if the execution of a migrating program is resumed at a destination site with exactly the same execution state as that of the migration time [Sek99]. A language that supports transparent migration usually provides a simple operation (like the **go** operation of Telescript[TV96], or **move to** of Nomadic Pict [Woj00]), that indicates at which point in the code the migration should occur. All low level operations like packing, marshaling and shipping of data are handled by the RTS of the language.

Systems that do not support transparent migration usually use mechanisms like *remote evaluation* to express mobility. Remote evaluation [Vol96] is an extension to RPC [Sri95] that allows the execution of code in a remote computer. A remote evaluation can be expressed like this:

$y :=$ **at** location **eval** $f(x)$

This request evaluates the function application $f(x)$ on the remote host `location`. There are several variants of remote evaluation that are implemented in many different languages, e.g., Kali [CJK95], GdH [PTL00] and the Tube System [Hal97].

Mobile systems implemented using explicit mobility differ significantly from traditional parallel and distributed systems, as can be seen in table 5.6. In parallel systems there is usually a static set of locations where a program is executed and the only objective is to speed up the execution of a *single stateless program*. The code for the application is already present in all the locations and no code is exchanged between

Table 2.1: Parallel, Distributed and Mobile Systems

	Locations	What is Migrated	Objective
Parallel	Static	Data	Speed up of a single program
Distributed	Dynamic	Data and Control	Client/Server applications
Mobile	Dynamic	Data, Control and Code	Distributed applications where there is no distinction between clients and servers

locations, there is only *data mobility* i.e., simple values like numbers and strings can be communicated. Distributed systems support the interaction of *different programs* running on different locations. The set of locations is *dynamic*: programs can join and leave the computation at any time. Computations are *stateful* as they have to interact with resources available at servers. Besides data mobility, distributed systems also provide *control mobility* [Car97], e.g., using RPC when a thread of control executing at one node calls a remote procedure, the control is passed to another thread at some other location. When the remote thread resumes execution, the control returns to the first location. No code is moved, only control and the data that is the argument for the remote procedure. Mobile computation on the other hand, is based on the movement of code and not the execution of code already available on remote locations. The code is moved together with control and the data representing the state of the computation. Another difference is that in distributed applications usually there are *servers* that provide services, and *clients* that connect to servers to request services. In mobile applications there is no distinction between clients and servers. A mobile computation that implements a service can migrate in order to use resources on remote locations.

2.4.1 Characteristics of Languages for Mobile Computation

Mobile Computation is a fairly new area, but it seems that researchers already agree that a language that supports mobile computation should have at least the following properties [Kna95, CPV97, FPV98, Kir01, Car99]:

- *Location-aware*: In a mobile language, code mobility usually occurs when the program needs to access non-local resources. Thus, the programmer must be able to explicitly say to *where* the computation must be moved. This notion of locality does not need to be restricted to the name of machines in a network, for example, it could be related to the names of the *resources* that the programmer wants to

access. The notion of locality is important in a mobile language because interacting with local resources in most cases is completely different from interacting with remote ones [CPV97].

- *Mobility Primitives*: Mobility of code and computation must be under the programmer's control [FPV98]. The language must provide abstractions and mechanisms for programmers to indicate when computations should be migrated. This property is closely related to the previous one. If a program is aware of its location and it must move to use non-local resources, the programmer must have the power to specify in which cases computations and code should migrate to new locations.

- *Code mobility is exploited on an Internet-Scale* [FPV98]: Mobile languages should operate in large scale systems where networks are composed of heterogeneous hosts and it should support *open systems* in which multiple executing programs interact using a predefined protocol, as opposed to *closed systems* where there is a static set of programs and locations [TLP02].

We can also identify some implementation properties of particular importance for mobile computation:

- *Architecture Neutral*: As mentioned before, mobile code is beneficial for large-scale distributed systems. Mobile programs, like agents, can work as a common language in heterogeneous systems [Kna95], provided that all systems can interpret the primitives of the language. Hence the runtime system of the language must provide functionalities that abstract over architecture specific code, like data marshaling and the use a portable format of executable code, e.g., an architecture neutral byte code.

- *Security*: One of the main aims of mobile computation is to share resources among the participants of a distributed system. In order to obtain the advantages of mobile computation *safety* (received computation should not be the cause of errors that may prevent the program to present its expected behaviour) and *security* (against malicious code) must be considered. These languages must in some way identify and restrict the execution of programs which are potentially dangerous [Kir01].

2.4.2 Advantages of Mobile Languages

Mobile computation offers some advantages over traditional programming paradigms for distributed programming, overcoming some of the limitations of the client/server approach. Here we present some of these advantages [MDW99, FPV98]:

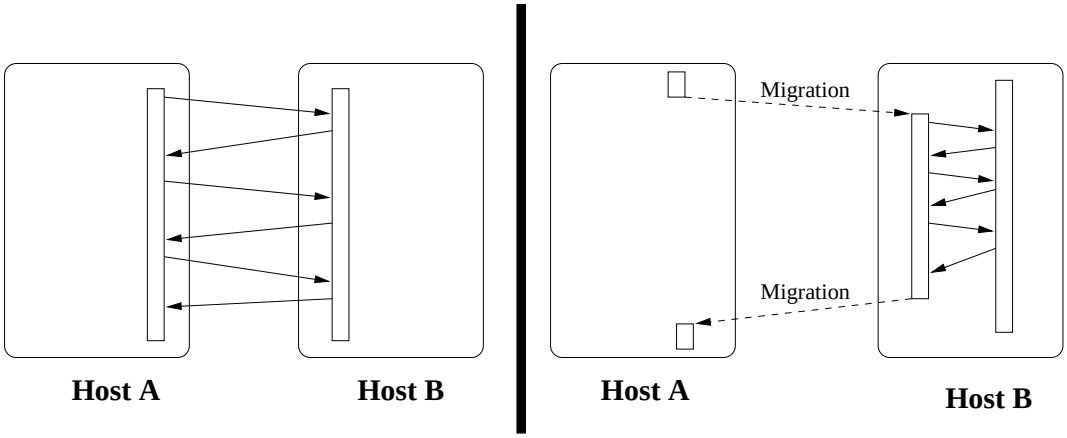


Figure 2.1: Network load reduction with mobile computation

- *Overcoming limitations of a client computer*: limitations such as memory size, processing power, low-bandwidth and storage may be overcome by sending computations to be executed remotely. Mobile computation can also help in reducing network traffic as many remote accesses to resources can be turn into local accesses plus the migration (as can be seen in Figure 2.1). For example, sending a computation to a database server in some cases will generate less traffic than making lots of remote queries.

- *Customisation*: It is very hard to modify a client-server architecture. Servers usually provide a fixed set of functionalities through which clients can interact with it. Often, clients may need extra functionalities and the only way of making them available is to upgrade the server with the new interface needed. With mobile computation, the server interface can be kept to a minimum set of core functionalities that seldom need to be modified. Clients can then send code to the server that uses this interface in order to provide the services they need. In the same way, mobile computation can help in maintaining and extending distributed software. In a distributed application, when a new functionality needs to be added, the new software must be installed or patched at each site. This probably

involves stopping the system and human intervention. Using mobile code, these new functionalities can be added on demand, once they are needed by the sites.

- *Adaptability*: Mobile computation can have the autonomy needed in order to adapt to changes in the heterogeneous environment to which the nodes of a network may be connected. For example, a mobile agent can identify and change its behaviour in order to cope with broken or low-bandwidth links.

- *Inherent survivability*: As a mobile computation can carry both state and the code of the action it is performing, it has a higher-degree of fault tolerance as a result [FPV98]. When a partial failure is detected (e.g., disk failure), the computation can move to another location where it resumes execution. Furthermore, in a traditional client-server architecture, the state of the computation is distributed between clients and servers, while in a mobile language the state is always carried by the mobile computation, making it easier to recover from failures locally as there is no need for knowledge of the global state.

- *Representing a disconnected user*: A user can send a computation to do some work inside a network and then disconnect, and collect the results later when he is connected again. This is one of the advantages that mobile computation provides for the users of mobile computing devices.

- *A new paradigm for development of distributed systems*: Making an analogy to the real world, many applications can be expressed using the idea of entities that move in order to achieve some goal. For example, a travelling salesmen that visits customers or a buyer who visits stores in order to find cheap prices, can all be represented as computations that visit hosts in the network in order to achieve some goal.

There are also some problems in the adoption of mobile code technologies. Sometimes migrating computations can be very expensive, e.g., if the objective of the computation is just to communicate a simple value like an integer. In some mobile agent languages an agent can not open network connections with remote hosts, if it wants to communicate a value, it must migrate to the remote host where the value is then delivered.

The main obstacle to the acceptance of mobile computation for commercial applications is security [Car97]. As discussed in section 2.4.1, a real world implementation of a mobile language should provide *safety* and *security*

2.5 Calculi for Mobility

Much research has been done on developing calculi to specify different aspects of mobility. Here we present the π-calculus one of the first of its kind, that is a mathematical model of the changing connectivity of interactive systems [Mil99]. Next we discuss some extensions to π-calculus that address different aspects of mobility.

2.5.1 π-Calculus

The π-Calculus [Mil99], is a process algebra in which processes interact by sending communication links to each other. The process that receives a link can then use this link to communicate with other processes.

The calculus has two kinds of entities - concurrent *processes* (P and Q in Figure 2.2) and communication *channels*. Channels are identified by globally unique names, that can be freshly created and communicated between processes. A process that receives a channel can then use this channel for communication.

In figure 2.2 we present the syntax of a simple asynchronous version of the π-Calculus. The term () represents an inactive process. $P|Q$ represents two concurrently

$P, Q ::=$	$()$	nil
	$P \mid Q$	parallel composition of P and Q
	$x!v$	output v on channel x
	$x?p \rightarrow P$	input from channel x
	$x? * p \rightarrow P$	replicated input from channel x
	new x **in** P	new channel name creation

Figure 2.2: Syntax of the π-Calculus

active processes. A process term $x!v$ sends the name v on channel x. The term $x?p \rightarrow P$ waits to receive a name on x, then substitutes p in P with this name after reception, and continues with P. We write $\{a/p\}P$ for the process term obtained by replacing all free occurrences of p in P by a. Using **new** x **in** P ensures that x is a new channel in P.

A replicated input $x?*p \rightarrow P$ is used to represent a server that after reading a value from x is ready to accept a new input; it behaves as an arbitrary number of parallel copies of $x?p \rightarrow P$.

The π-calculus models important issues of distributed systems such as concurrency and communication. In early process algebras like CCS [Mil89] and CSP [Hoa85] the model of communication relies on a static connection between processes. The π-calculus is based on the idea of naming and communication of channel names. As processes can create new names, and communicate these names to other processes, the model reflects the dynamic nature of the communication between processes/threads in a concurrent system. The π-calculus is a model for concurrency, but it also models the message-passing paradigm for network communication. Processes can be seen as running programs connected in a physical network represented by the channels. In a network using Internet protocols, programs can only communicate if they know the the IP and Port number of the destination, while in the π-calculus, processes communicate if they know the name of a channel in which the destination is waiting for messages.

The π-calculus is a *first-order* calculus, meaning that mobility is achieved by allowing transmission of names (*name mobility*), but not arbitrary terms. One of the main reasons for this design choice is the belief that the communication of names is enough to model communication involving processes, as described in [San01]. Furthermore, the mobility of links as provided by the π-calculus is expressive enough to simulate mobility of processes, as explained in [Mil99].

The π-calculus, besides being a model of mobility, is also a basic model of computation, based on the notion of *interaction* [Mil99], in the same sense as the λ-calculus [Bar84] is a basic model of computation based on mathematical functions.

2.5.2 Other models for mobility

A natural extension to the basic π-calculus would be to allow other objects besides names to be communicated through channels, as for example tuples in the *polyadic* π-calculus [Mil93].

One important issue for distributed computing that is not addressed by the π-calculus is the notion of locality, or physical distribution of processes on different sites. Many variants of the π-calculus were proposed in order to study different aspects of distributed communication, here we present some of them:

- Distributed-π [Sew98], adds the notion of channel location and process migration from the distributed join calculus [FGL+96], to the asynchronous π-calculus. It is used to study how locality restrictions on the use of capabilities can be enforced by a static type system.

- Nomadic-π [Uny01], is used to study a distributed infrastructure for mobile agents. The calculus has *first-order* channels and primitives for the migration of agents. This calculus is the basis for a programming language called Nomadic-Pict [Woj00], which is discussed in more detail in Chapter 4.

- The Ambient Calculus by Cardelli and Gordon [Car99], is used to model mobility between different security domains. An ambient is a bounded location that may contain processes and other Ambients. An ambient can move as a whole, in and out of other ambients.

- The Distributed Join Calculus [FGL+96], extends the Join Calculus [FG96] with locations and primitives for mobility. Locations in the calculus can have processes and sublocations, thus locations, as ambients in the ambient-calculus, can be seen as a tree structure. When a location moves to another site, it takes with it its sub-locations. The Distributed join calculus is the basis for the Jocaml [CF99] language, discussed in more detail in Chapter 4.

2.6 Summary

In this chapter we reviewed early work on mobility and presented the main concepts of mobile computation. In the mobile computation paradigm a software that starts its execution in one location in a network can halt its execution, and migrate to another location where it continues to execute. Mobile computation can be *implicit*, where computations are automatically migrated by the system, e.g., runtime system of the language, for better load balancing of tasks, or *explicit* where a mobile language provides primitives that the programmer can use to choose when migration will occur.

There are many advantages for using mobile computation technology. Mobile computation helps sharing resources available in both small and large scale networks, and it also supports a more flexible form of distributed systems where non-local computations do not need to be known in advance at the execution site [Tho97]. Mobile computation

can also, depending on the application scenario, reduce network load by moving the code closer to the data it uses.

There has been a lot of work in defining a formal foundation for mobile computation and many calculi based on the π-calculus were designed for this purpose.

In the next Chapters the concepts presented in this background Chapter are applied in the design of purely functional mobile languages.

Chapter 3

Thread Migration in a Parallel Graph Reducer

The potential of functional languages to support parallelism with minimal programmer intervention has been long recognised [Weg71], but has only recently been realised using sophisticated language implementations, e.g., [HP90, THM+96, BLOMP97]. In purely functional languages, subexpressions of a given term can be evaluated at any order without the risk of interference [JH93], and this is exploited in many parallel functional languages, e.g., Haskell [TLP02]. Therefore, in a language with *implicit* mobility, running threads can be reallocated to improve processor utilisation without affecting the result of the computation.

Parallel functional languages typically generate massive, but fine-grained, parallelism and an implementation must have effective mechanisms to distribute work across the parallel machine. In many models potential work is easily and cheaply distributed, e.g., in graph reduction a *spark* is simply a reference to an unevaluated closure in the graph. On receiving a potential work item an idle processor will create a *thread* to perform the computation. Threads are more heavy weight than sparks as they have an execution state, typically including stack(s) and a set of registers.

The performance of certain classes of programs can be improved if, in addition to potential work, threads can be distributed. These are programs with poor load balance leading to under-utilisation of some processors: some processors are idle while others have several threads to execute. Many data parallel programs are vulnerable to this, especially those that generate parallelism only at the start of execution, e.g., there are

reports of poor load balance in large-scale programs written in the parallel functional language GpH [LTH+99].

As a simple example, Figure 3.3 shows an overall activity profile of the sumEuler program, written in GpH, discussed in section 3.4. The profile is recorded on 8 processors and shows execution time on the X-axis and the number of threads on the Y-axis. The shades of gray in the figure represent the different states of the threads, and the key states are *running*, i.e., currently executing and *runnable*, i.e., could be executed but residing on a processor currently executing another thread. For much of the execution there are idle processors and runnable threads simultaneously. If these threads can be *migrated* from a heavily-loaded processor to a lightly-loaded processor, runtime can be reduced.

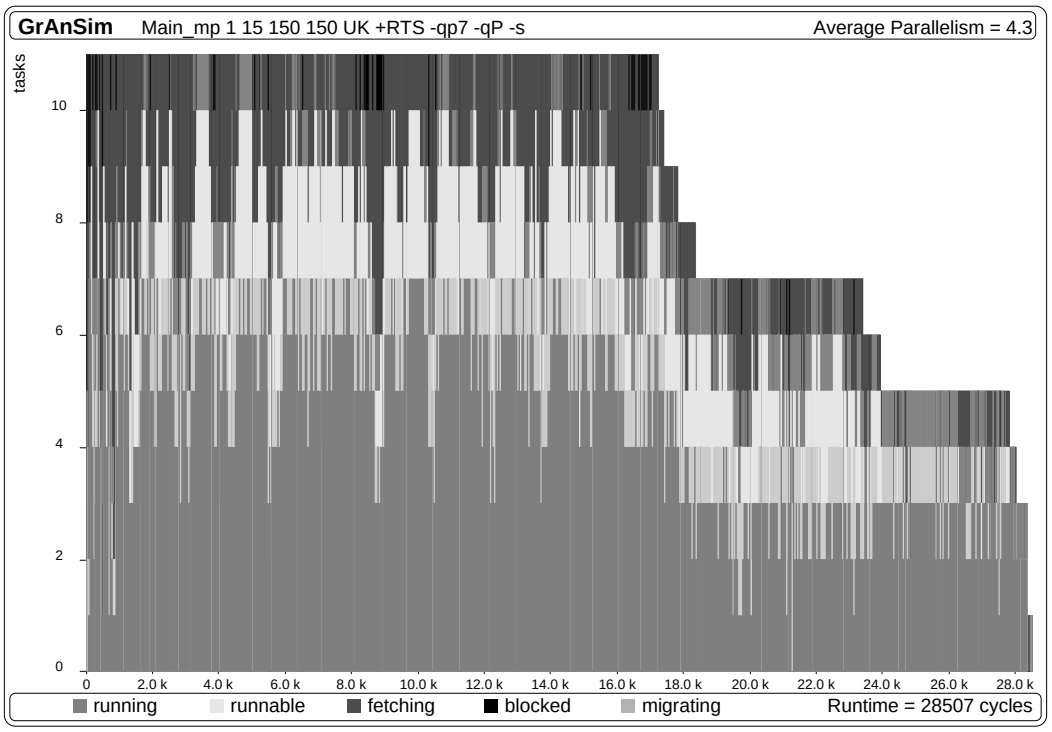


Figure 3.3: Activity Profile of sumEuler, a Program with Migratable Threads

Although conceptually simple, engineering effective thread migration in a sophisticated compiled language is challenging. The execution state of a thread has an elaborate representation that must be carefully packed, communicated and unpacked for execution to resume. Moreover the sharing of data and computations by the thread with other threads on the source and destination processors must be preserved. A

further consequence is that migrating threads is not only expensive but also destroys any locality of reference the thread enjoyed on the source processor. In consequence thread migration must be carefully scheduled, to be used only when other load balancing mechanisms have failed. It is salutary that relatively few systems have been constructed, although examples include Cilk [BJL+95] and the Grip System [HP92]. This Chapter describes the implementation of thread migration in the GUM runtime system [THM+96] supporting GpH(*Glasgow Parallel Haskell*) [THLP98].

The remainder of the Chapter is structured as follows. Section 3.1 describes GpH, and its associated GUM runtime system, including existing work distribution mechanisms, is described in Section 3.2. Section 3.3 presents the design and implementation of the new thread migration mechanism. Section 3.4 gives preliminary performance measurements. Section 3.5 covers related work and Section 3.6 concludes.

3.1 GpH

GpH(*Glasgow Parallel Haskell*) [THLP98] (see Figure 3.4) is an extension of Haskell 98, using the parallel combinator `par` to specify parallel evaluation, and `seq` for sequential composition. The expression `p `par` e` (here we use Haskell's infix operator notation) has the same value as `e`. Its dynamic effect is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing the evaluation of `e`. As `p` and `e` may share common variables, anything evaluated in parallel in `p` can be returned via `e`. The `par` constructor only identifies expressions that could be evaluated in parallel, but the runtime-system is free to ignore any available parallelism. The `seq` combinator forces the evaluation of its first argument to WHNF (*Weak Head Normal Form*), before returning its second argument as a result.

Higher-level coordination is provided using *evaluation strategies*: higher-order polymorphic functions that use `par` and `seq` combinators to introduce and control parallelism [THLP98]. In Figure 3.4 we see the basic operations for strategies. The `using` function applies a strategy to an expression. The basic strategy is `whnf`, which evaluates a value to WHNF, the default evaluation in Haskell. It is implemented as a simple application of the `seq` operator. The `rnf` strategy evaluate values to normal form, and is instantiated for all major types.

As an example using strategies, in Figure 3.4, `parMap` applies the function `f` to

```
par :: a -> b -> b          -- parallel composition
seq :: a -> b -> b          -- sequential composition

using :: a -> Strategy a -> a
using x s = s x `seq` x     -- strategy application

rwhnf :: Strategy a         -- reduction to whnf

class NFData a where        -- class of reducible types
      rnf :: Strategy a     -- reduction to normal form

parMap f xs = map f xs `using` parList rnf -- parallel map
```

Figure 3.4: GpHprimitives and Evaluation Strategies

all the elements of the list `xs` in parallel. This parallel version of the `map` function is implemented using the `parList` and `rnf` strategies. The `parList` function evaluates the elements of a list in parallel to the degree specified by its argument, in this case, to normal form using the `rnf` strategy. The `parList` and `rnf` strategies have a straightforward implementation using `par` and `seq`.

3.2 The GUM Virtual Machine

3.2.1 Introduction

GUM (Graph reduction for a Unified Machine model) [THM+96] is a distributed runtime system implemented as an extension of GHC's (Glasgow Haskell Compiler) [Mar05] sequential RTS. It is the core of the implementation of a collection of parallel/distributed Haskell extensions such as GpH [THLP98], Eden [BLOMP97] and GdH [PTL00]. GUM has been ported to many different architectures (both shared and distributed memory) and using many different message passing libraries (i.e., PVM [GBD+94], MPI [GLS99] and Globus [FK99]). The version of GUM used for the experiments on thread migration uses the PVM library for communication and runs on a Beowulf cluster [RBMS97], i.e., a dedicated cluster of workstations, often interconnected with a high speed network.

The main characteristics of GUM are:

- *Distributed Shared Heap*: parallel reduction of a program evaluates the graph that

represents the program in memory. All processing elements (PEs) share a single graph stored in a global shared heap. To avoid duplicating reduction, all graph sharing is preserved. The distributed shared heap provides transparent access to heap addresses, i.e., local and remote addresses are accessed in the same way.

- *Lazy task distribution*: Work is never exported eagerly to processors for load balancing. Instead, when a processor becomes idle, it searches work in the *spark pools* of other PEs. A spark pool is a data structure that contains potential work in the form of pointers to expressions that could be evaluated in parallel.

When the programmer uses the `par` combinator in the code to indicate that a value could be evaluated in parallel, a pointer to the graph representing this value is added to the *spark pool*. A spark only represents potential parallelism. When a processor is idle, it looks for a spark in the spark pool and if successful this spark is turned into a *thread* that will evaluate the *thunk* (unevaluated closure). The problem with this approach for load distribution is that in some cases, at the end of the execution, when there are no more sparks in the spark pool, some processors are idle while others have several threads to execute. Many data parallel programs are vulnerable to this, especially those that generate parallelism only at the start of execution. This problem could be solved if in addition to potential work, threads could be distributed.

3.2.2 The GUM Runtime System

This section gives an overview of the GUM distributed virtual machine, discussing its main components and focusing on design and importance of thread migration for load balancing.

Figure 3.5 summarises the main components of GUM. Potential parallelism is represented as sparks, i.e., pointers to graph structures in the heap, which are collected in a spark pool. Sparks are generated by executing the `par` primitive. Threads are created on a processing element (PE), consisting of a CPU and local memory, if it is idle, i.e., if its thread pool is empty. More threads are added from blocking queues when the required data becomes available. By design the generation of sparks is very cheap, consisting only of adding a pointer to the thunk (unevaluated closure) to the PE's spark pool, while threads are far more heavy-weight, although still light compared to usual OS threads. Both spark and thread pools are managed as FIFO

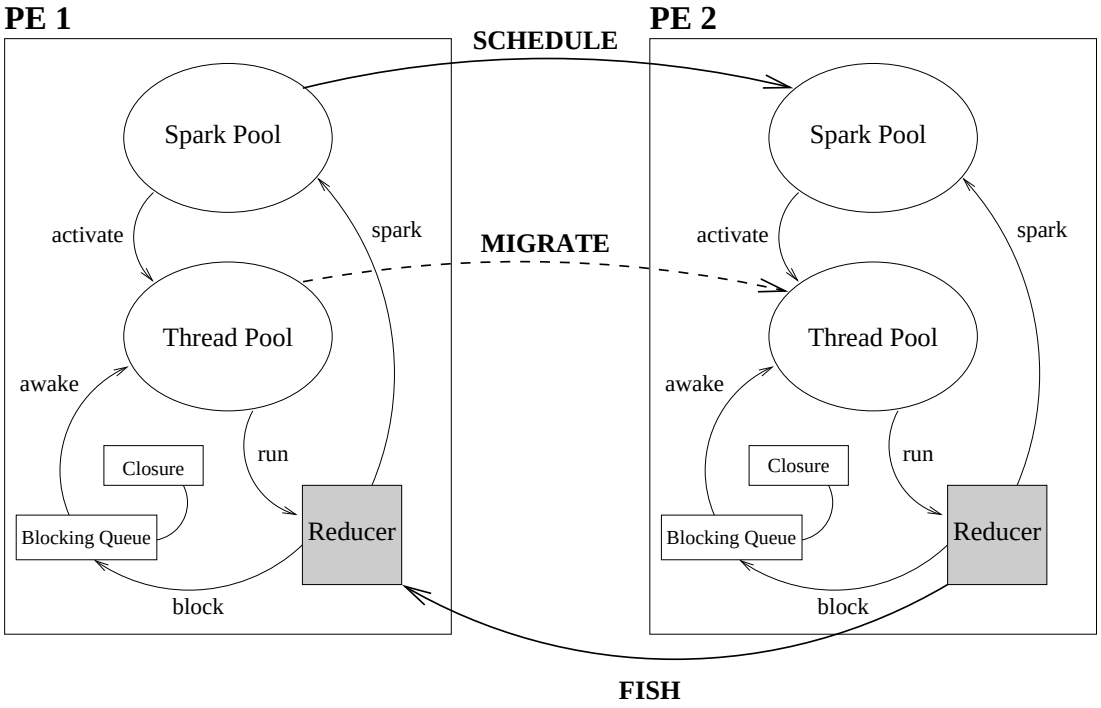


Figure 3.5: Interaction of the Components of a GUM PE.

queues. To create a new thread the PE activates one of its sparks, generating a Thread State Object (TSO), which holds essential information such as registers and a stack pointer. The main component of GUM is the graph reduction engine (called Reducer in the Figure) which besides reducing the graph (program) being evaluated, it also contains the Scheduler that controls all other parts of GUM by issuing the messages (run/activate/block/awake/spark) that appear in Figure 3.5. The scheduler of each PE chooses one thread from the thread pool to run on the graph reducer. If a running thread blocks on unavailable data, it is added to the blocking queue of that node. A blocked thread is added to the thread pool when the required data becomes available, either because a local thread produces it or the data arrives from another processor.

Figure 3.6 illustrates the communications induced by the existing spark distribution mechanism, a *work stealing* scheme. If the Scheduler does not find a thread or a spark to execute, it sends a FISH message requesting work. The FISH message specifies the PE requesting work and also contains an age limit, i.e., the maximum number of PEs to visit, in the form `FISH(Source, Age)`. Initially all PEs except for the main PE are idle and without sparks. Idle PEs send a `FISH` message to a PE chosen at random, and only ever have one outstanding FISH. In the example, PE A sends a FISH to PE C. If a FISH recipient has an empty spark pool it increases the age and forwards the FISH

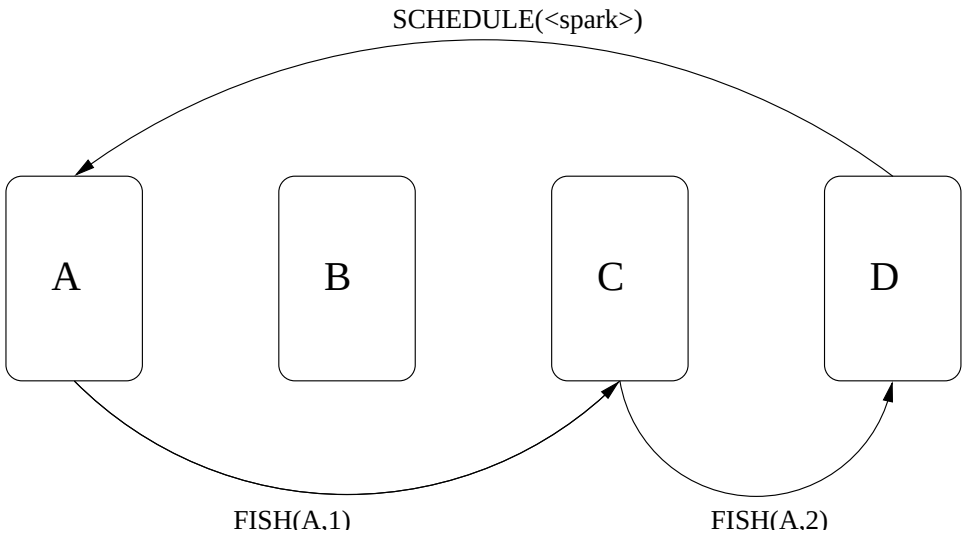


Figure 3.6: Spark Distribution in GUM

to another PE chosen at random, in our case PE D. If a FISH recipient has a spark it sends it to the source PE as a SCHEDULE message: in the example, PE D sends to PE A. The age limit of a FISH is used to avoid swamping a lightly loaded machine with FISH messages: if the age reaches the limit (a tunable system parameter) the FISH is returned to the source PE which delays before reissuing another FISH.

3.2.3 Memory Management

Parallel programs are represented in the heap as a graph that can be evaluated concurrently using the processors available. To avoid duplication of work, GUM provides the abstraction of a global heap that is shared by all the PEs in the distributed system, as depicted in Figure 3.7. Every PE has a local memory, that is part of the distributed heap, and there is a two-level addressing scheme distinguishing *local addresses* (LAs) from *global addresses* (GAs), that reference values in the shared heap. Global Addresses are managed so that each PE can garbage collect locally, without having to synchronise with other PEs.

A GA is created when GUM's work stealing mechanism forces a PE to send work to another PE: after the *thunk* is sent to its destination, it is overwritten in the original PE with a *FetchMe* closure that contains the GA of the thunk on the destination. The original thunk is overwritten with a FetchMe to indicate that it is being evaluated in some other PE, and to indicate its new location, in case the original PE needs the result of its evaluation in the future. The GA contains an immutable identifier and a

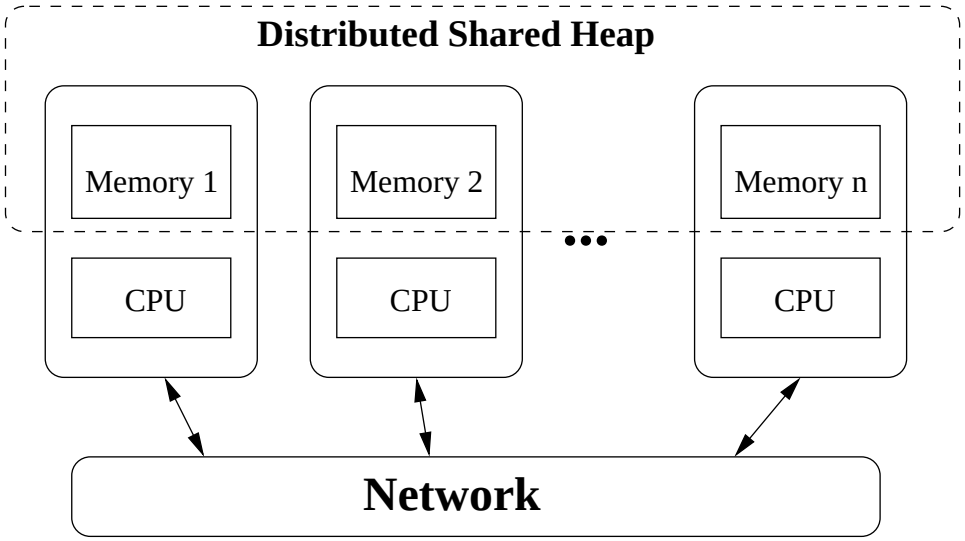


Figure 3.7: Distributed Shared Heap

PE identifier, and each PE contains a GIT (Global Indirection Table), that maps the identifiers to the local address in the local heap. The GIT is used as a source of roots for local garbage collection, and after garbage collection it is adjusted to reflect the new addresses of closures in the local heap. Weighted reference counting is used to recover local identifiers and GAs during garbage collection. GUM's addressing mechanism and distributed garbage collector are described in detail in [THM+96].

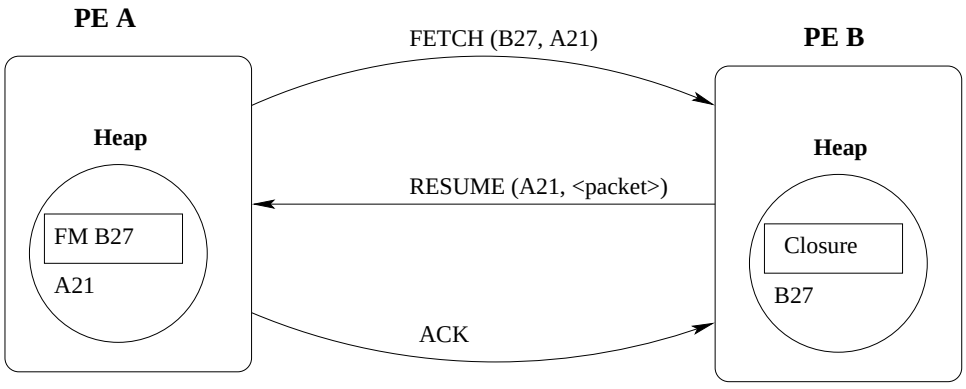


Figure 3.8: Fetching Graph

When a thread attempts to evaluate a FetchMe closure, the FetchMe is converted into a *FetchMe blocking queue*, and the thread is blocked and added to the queue. Any other thread trying to enter the FetchMe will also be enqueued. Then a FETCH message is sent to the PE that owns the GA in the FetchMe. In Figure 3.8, PE A sends a FETCH message requesting the graph in the address B27. The FETCH message also

contains the new GA for B27, which is the address of the FetchMe. When receiving a FETCH message, the PE will pack the appropriate closure in a packet and send it back in a RESUME message. When the RESUME arrives, the graph is unpacked, the Fetch-Me is redirected to point to the root of the unpacked closure, and all threads blocked in the queue are awaken. After that, an ACK message is sent to confirm the new location of the closure.

3.2.4 Graph Packing

To communicate a closure, the graph representing this closure must be packet, or serialised, i.e., converted into an array of bytes that can be easily communicated. Packing just a single closure in messages may be too expensive in high-latency networks. When a closure is packed, some "near by" reachable graph is added to the packet in order to reduce the number of FETCH messages that need to be sent, i.e., the graph is packed breadth-first, up to a fixed limit.

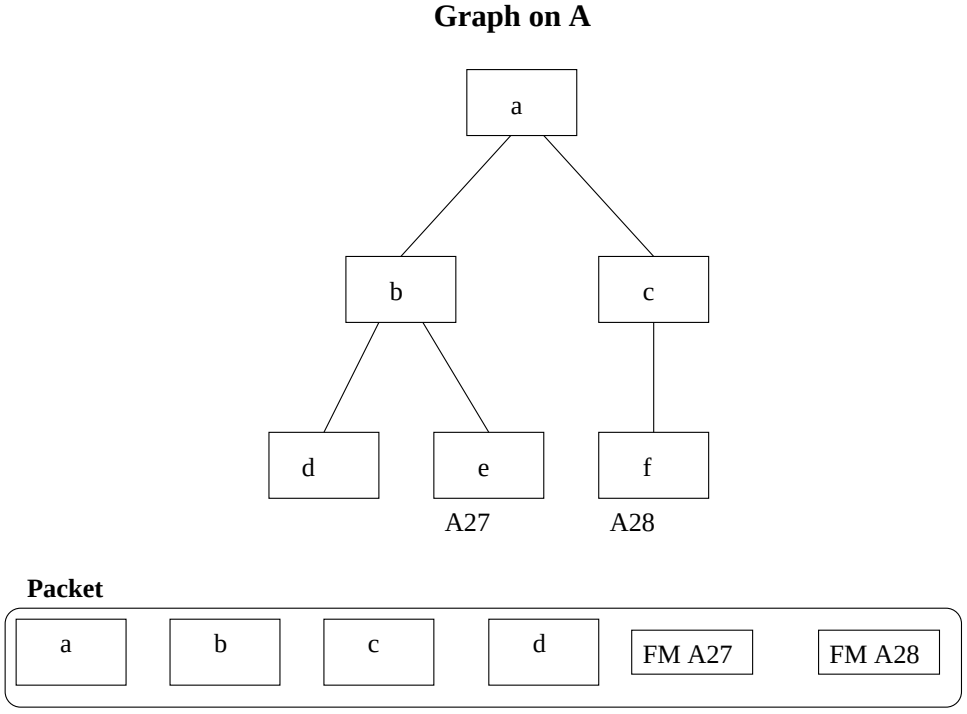


Figure 3.9: Graph Packing

While a graph is being packed the addresses of the closures are kept in a temporary table so that sharing and cycles can be detected. The graph is packed up until the packet is full or all the reachable graph is already packed. If there is no space left

in the packet, Fetch-Me closures and GAs are used to reference the graph that is left behind (Figure 3.9). Each closure packed is made global, but a few closures are packed specially:

- Values that are already in normal form are not globalised, as they are already evaluated, they can be freely copied between machines.
- *Black Holes*, i.e., closures that are already being evaluated, are packed as Fetch-Mes to the black hole.

The algorithm for unpacking reads the packet and reconstructs the graph breadth-first.

3.3 Implementing Thread Migration

3.3.1 Scheduler

A central component in GUM is the scheduler, available at each PE, which determines which thread to execute next. To implement thread migration the following scheduling policy is used to ensure that thread migration is attempted only if other cheaper work location schemes fail. The new policy is an extension of the existing policy and has been tested for simulated parallel execution [LH96], but hasn't previously been available in GUM.

1. execute another runnable thread, if available;
2. turn a spark into a thread if no runnable threads are available;
3. try to acquire a remote spark if the processor has no local sparks;
4. *try to* ***migrate*** *another runnable thread if no remote sparks can be found*;

Several scheduling alternatives are possible. Currently the TSO of a migrated thread is added at the end of the (FIFO) runnable queue. Migrated threads could be preferred by inserting at the front of the runnable queue, or more generally by distinguishing them in the queue, by partitioning the queue into priority-based regions. Realistically, when a PE asks for a remote thread, it is because it did not have anything else to execute (i.e., neither sparks or other runnable threads), hence adding the new thread to the end of the queue seems the appropriate alternative.

3.3.2 Communication Protocol

The communication protocol in GUM, as described before, is very simple and consists of only 6 classes of messages [THM+96]. Thread migration introduces two new messages, both variants of existing messages: a SHARK which is a variant of the FISH message; and MIGRATE which is a variant of the SCHEDULE message and transmits a thread and an associated graph structure between PEs.

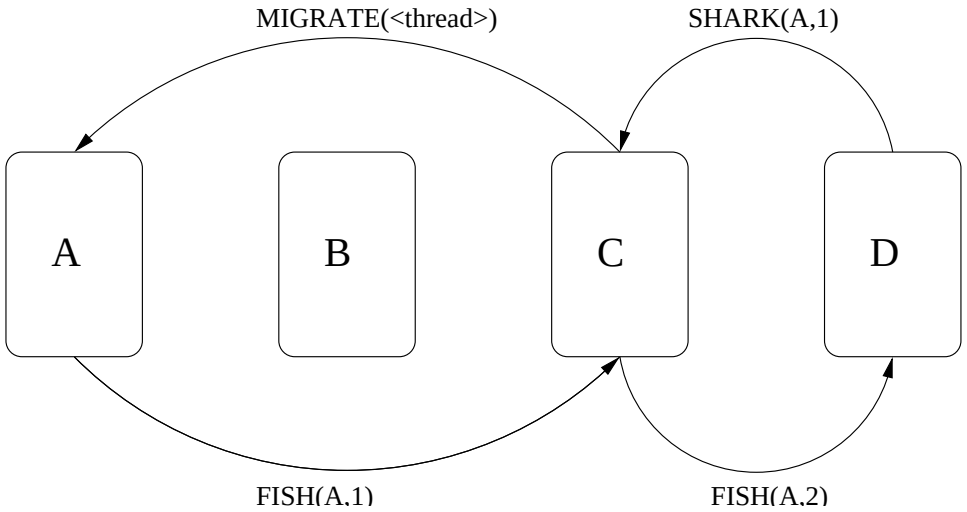


Figure 3.10: Thread Migration in GUM

Figure 3.10 illustrates the communications induced by the new thread migration mechanism. As before, a FISH seeks to locate a spark, and in our example the FISH visits PEs C and D unsuccessfully. Now, however, when a FISH reaches its age limit instead of returning to its source PE, it becomes a SHARK message with age 1 and is forwarded to a random PE. When a SHARK arrives, if the PE has a spark it is sent to the source PE in a SCHEDULE message; otherwise if the PE has a runnable thread it is sent to the source PE in a MIGRATE message; otherwise the PE increases the age and forwards the SHARK at random. SHARKs and FISHes have the same age limit, and a SHARK reaching this age limit is returned to the source PE. In the example the SHARK finds a thread, but no spark on PE C, and migrates it to PE A.

At the moment GUM does not propagate load information between PEs. One potential improvement of the load balancing mechanism would be to carry information about spark and thread pool sizes of the visited PEs as part of a FISH message. In our experience even the naive, but cheap mechanism of randomly choosing the target of a FISH works well for most applications. In the presence of thread migration the

additional overhead might be justified, though, because in general runnable threads are much rarer than available sparks. Similarly, information about the granularity of threads would be useful in order to choose the largest thread to migrate. However, such information is not directly available and hard to obtain automatically [Loi98].

3.3.3 Thread Packing

The main modification to enable thread migration is to provide mechanisms to pack and unpack threads for communication between PEs. This entails packing a TSO and its associated stack. Since a TSO is a (slightly special) heap object, we extend the cases for packing a graph node, to include a case for a TSO. Packing most of the entries in the TSO is uncomplicated, since they are static data rather than pointers. The exceptions are the pointers in the stack that have to be adjusted when unpacking the TSO.

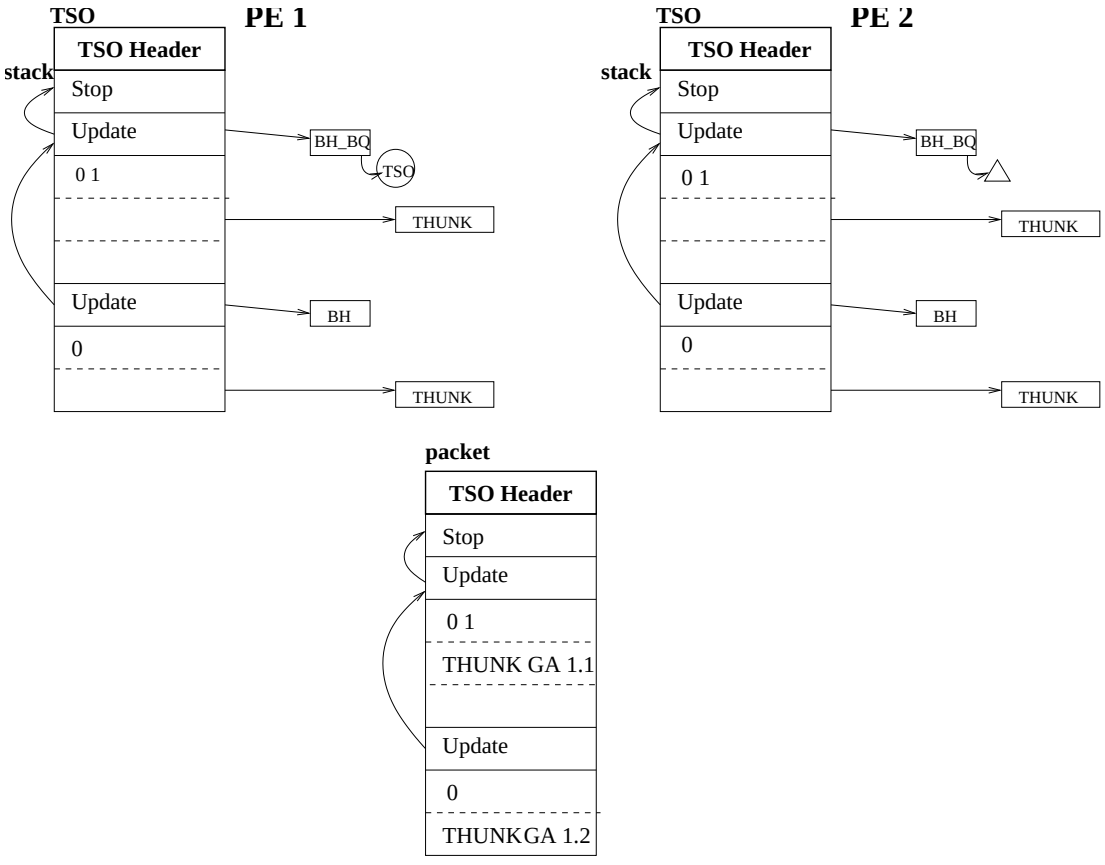


Figure 3.11: Transfer of a Thread (TSO) between PEs.

GUM is a parallel extension of the STG-machine [PJ92a] and the TSO and stack layout is unchanged. Figure 3.11 summarises the transfer of a thread, represented as a TSO, from PE 1 to PE 2 (note, that the stack grows downwards). The TSO can

be partitioned into a header, containing mostly non-pointer data, and the stack. The stack consists of a continuous sequence of variable sized activation records. Each record starts with one of 4 possible frame types: an update frame, a stop frame, a seq frame or an exception frame. The most common type is an update frame, which contains a pointer into the heap, pointing to a graph structure that will be updated with the result of the current evaluation. As shown in Figure 3.11, the type of the updatee will be either BH, a "black hole" representing a graph structure under evaluation, or a BH_BQ, a "black hole blocking queue" which additionally contains a list of TSOs waiting for the result of this evaluation (this list is represented by the circle in Figure 3.11). An exception frame contains a pointer to the code that has to be executed when catching an exception. A seq frame contains a pointer to the code corresponding to the second part of a sequential composition (the `y` in a `x 'seq' y` construct). A stop frame can only occur at the bottom of the stack and indicates the end of the computation for this thread. All frames are linked together, with one thread register pointing to the top-of-stack frame. The layout of an activation record itself is specified by a bitmask immediately after the update frame, with 1 representing data and 0 a pointer entry. Since this layout is similar to the one of a partial application closure we can treat the elements of one activation record on the stack in the same way as the available arguments in a partially applied function when packing the stack.

During packing, the overall structure of the stack is maintained, but some modifications are made. As with all graph structures, global addresses (GAs) have to be allocated for pointers, in order to ensure that thunks, or unevaluated closures, are uniquely identified in the virtual shared heap. This can be seen in the packet in Figure 3.11 where the two thunks are packed with new global addresses GA 1.1 and GA 1.2. As with ordinary graph structures, a mapping of old GAs on the source PE, to new GAs on the target PE, is sent back as a reply to the communication shown here and the BH and BH_BQ closures on PE 1 are converted into FetchMe and FetchMe_BQ (Blocking Queue) closures. If a thread on PE 1 demands a thunk being evaluated by the migrated TSO, a FETCH request will be sent to PE 2 upon entering the FetchMe or FetchMe_BQ closure.

Note that when unpacking the black hole blocking queue (BH_BQ) on PE 2, a different kind of closure has to be used to represent the TSO that is blocked on the black hole on PE 1 (represented by a triangle in Figure 3.11). This closure is a "blocked

Table 3.2: SumEuler Runtime(s) with/without Migration

	Mean		Min.		Max.		Avg No	% Range		Performance Improvement
Numb PEs	**Mig.**	No Mig.	**Mig.**	No Mig.	**Mig.**	No Mig.		**Mig.**	No Mig.	Mean Runtime
1	**97.5**	97.5	**97.3**	97.4	**97.7**	97.6	0	**0.4%**	0.2%	0%
2	**51.9**	52.5	**50.2**	50.1	**54.2**	55.4	1	**7.7%**	10%	1.1%
4	**23.5**	26.1	**20.5**	21.4	**26.3**	31.7	4.2	**24.6%**	39.4%	10%
6	**17.0**	23.7	**15.66**	20.4	**18.8**	27.3	3	**18.4%**	29.1%	28%
8	**14.7**	20.8	**12.8**	14.9	**17.8**	27.5	4.0	**34%**	60.5%	29%
10	**11.7**	17.7	**9.5**	14.7	**12.9**	20.2	3.4	**29%**	31%	33%
12	**10.6**	17.5	**9.0**	12.6	**11.9**	20.8	3.4	**27.3%**	46.8%	39%
14	**10.5**	15.1	**8.1**	11.8	**11.8**	19.7	4.6	**35.2%**	52.3%	30%
16	**11.2**	14.3	**9.8**	8.4	**12.5**	19.0	5.2	**24.1%**	74.1%	21%
18	**8.8**	13.5	**7.9**	11.2	**9.2**	14.8	3.4	**14.7%**	26.6%	34%
20	**8.4**	12.8	**5.3**	9.0	**11.1**	14.9	4	**69%**	46.3%	34%
22	**8.5**	13.1	**5.7**	10.9	**11.4**	17.6	4.6	**67%**	51%	35%

fetch" closure, which already exists in GUM. It normally represents a fetch request from another PE, and contains information about the requesting PE and TSO, so that upon updating the BH_BQ closure, a message with the result data is sent to the original PE. By this mechanism the TSO on PE 1 will continue as soon as the migrated TSO updates the BH_BQ on PE 2.

3.4 Performance Measurements

The measurements in this section are performed on a high-latency cluster: a Beowulf [RBMS97] consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128KB cache, 128MB of DRAM and 5.7GB of IDE disk. The workstations are connected through a 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2.

3.4.1 Performance Improvement

Experience has shown that many GpH programs have migratable threads and some under-utilised processors [LTH+99], and three programs from two parallel paradigms are discussed in this section.

The first program, sumEuler, is data parallel: it computes the sum of the Euler totient function over an integer list [LTB01]. Figure 3.12 shows mean speedup curves for sumEuler with and without migration calculated from five executions of the program.

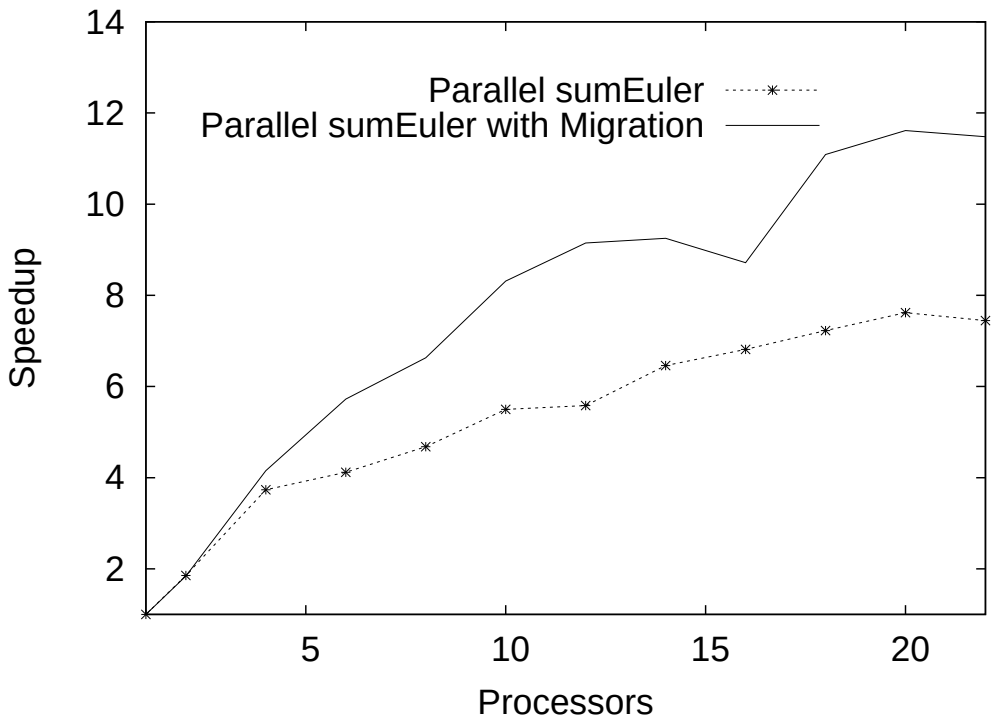


Figure 3.12: Speedups for sumEuler

The speedups reported here and throughout this Section are *relative*, i.e., improvement over the single-PE parallel execution. Table 3.2 shows, for varying numbers of PEs: the mean, minimum and maximum runtimes, the average number of threads migrated in each execution, the range of runtimes as a percentage of the mean runtime and the percentage reduction in mean runtime.

The sumEuler results show that thread migration improves runtime for all numbers of PEs measured. For small numbers of PEs the improvement is limited by the small numbers of migratable threads, but between 6 and 22 PEs the improvement is approximately 30% with a single exception. The improvement is variable, as discussed in the next section, with the greatest improvement being 39% on 12 PEs.

The second program, Maze, uses a divide-and-conquer algorithm to search a maze for an exit [DBPLT02]. Figure 3.13 shows the mean speedup curves and table 3.3 the performance improvements. The results show that thread migration improves performance on all numbers of PEs; from 4 PEs onwards improvements of approximately 13% are achieved with two exceptions. The improvement is variable with the greatest improvement being 21% on 16 PEs.

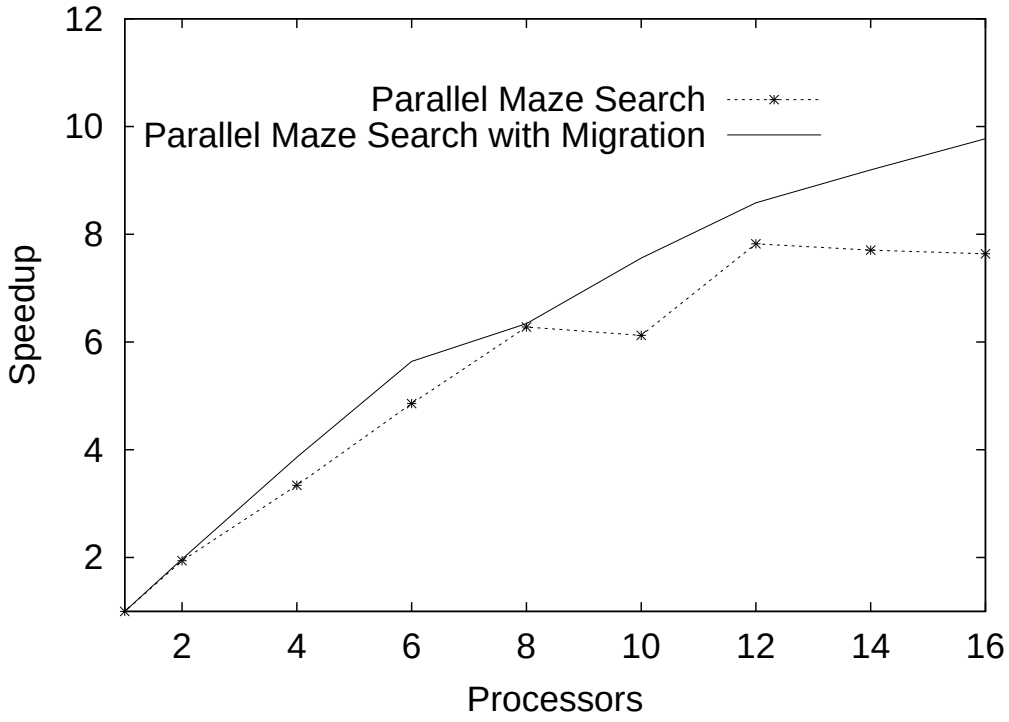


Figure 3.13: Speedups for Maze

The third program, Queens, is data parallel: placing chess pieces on a board. Figure 3.14 shows the mean speedup curves and table 3.4 the performance improvements. The results show that while thread migration improves performance on most configurations, it degrades it on two; the 4-PE and 8-PE configurations. There is an enormous amount of variability in the improvement, with a maximum of 37% and minimum of -10%.

Thread migration does not consistently or significantly improve the performance of either Queens or Maze up to 8 PEs because both have excellent processor utilisation, achieving a speedup of approximately 6 in each case. If the default GUM work distribution mechanism is achieving good utilisation, it is hard for the more expensive migration mechanism to improve on it. Indeed results for Queens show that the additional communication introduced may reduce performance (e.g., at 4 and 8 PEs), and increase variability. In contrast migration delivers significant and consistent improvements when utilisation is low, e.g., on 8 PEs sumEuler without migration has a speedup of just 4.7 and migration delivers a 29% improvement. In a similar way, migration improves the performance of both Queens and Maze at higher numbers of PEs as the utilisation delivered by the default load balancing mechanism falls.

Table 3.3: Maze Runtimes(s) with/without Migration

	Mean Runtime		Min. Runtime		Max. Runtime		Avg No Thr Mig	% Range		Performance Improvement
No PEs	**Mig.**	No Mig.	**Mig.**	No Mig.	**Mig.**	No Mig.		**Mig.**	No Mig.	Mean Runtime
1	**162.9**	162.0	**162.6**	159.7	**163.7**	163.5	0	**0.6%**	0 %	0%
2	**82.7**	83.5	**82.5**	83.3	**82.9**	83.5	1	**0.4%**	0.2%	0.9%
4	**42.2**	48.5	**40.4**	43.1	**44.1**	58.2	0.8	**8.7%**	31.1%	13%
6	**28.9**	33.3	**27.9**	30.8	**30.3**	35.1	2	**8.3%**	12.9 %	13%
8	**25.7**	25.8	**23.8**	24.9	**28.3**	29.1	5.8	**17.5%**	16.2%	0%
10	**21.5**	26.4	**19.8**	24.5	**24.1**	29.6	2.4	**20%**	19.3%	18%
12	**18.9**	20.7	**16.3**	16.7	**22.0**	24.6	4.2	**30.1%**	38.1%	8%
14	**17.7**	21.0	**16.0**	18.0	**20.3**	24.7	4	**24.2%**	31.9%	15%
16	**16.6**	21.2	**12.2**	16.4	**20.3**	25.3	4.4	**48.7%**	41.9%	21%

In summary, thread migration, in the examples, always improves the runtime of programs with migratable threads and low processor utilisation, i.e., idle PEs, like sumEuler and Maze. Thread migration often improves but may degrade programs with migratable threads and good processor utilisation, like Queens. In both cases the runtimes and improvements achieved are variable. The improvements are achieved by migrating a relatively small number of threads: typically around 4. This indicates that the migration policy described in section 3.3 works adequately, striking a balance between good data locality and even load distribution.

3.4.2 Variability

We hypothesised that thread migration would reduce the variability in runtimes, as it allows a poor initial distribution of sparks to PEs to be rectified. The '% Range' column in tables 3.2 and 3.3 show that migration reduces the range of performance results for programs with low utilisation, like sumEuler and Maze. However, table 3.4 shows that migration may increase the variability of some programs with good utilisation, like Queens on small numbers of processors. In these cases, we have observed increased communication in programs with migration and suspect the variability is due to both increased communication and potential blocking after having migrated threads. Depending on the amount of sharing with other graphs on the original PE, threads on a PE may send data requests and become blocked, thereby reducing the gain in performance due to migration. However, none of the programs with poor utilisation seems to suffer from an increased amount of data transfer caused by migration. Hence, it is important

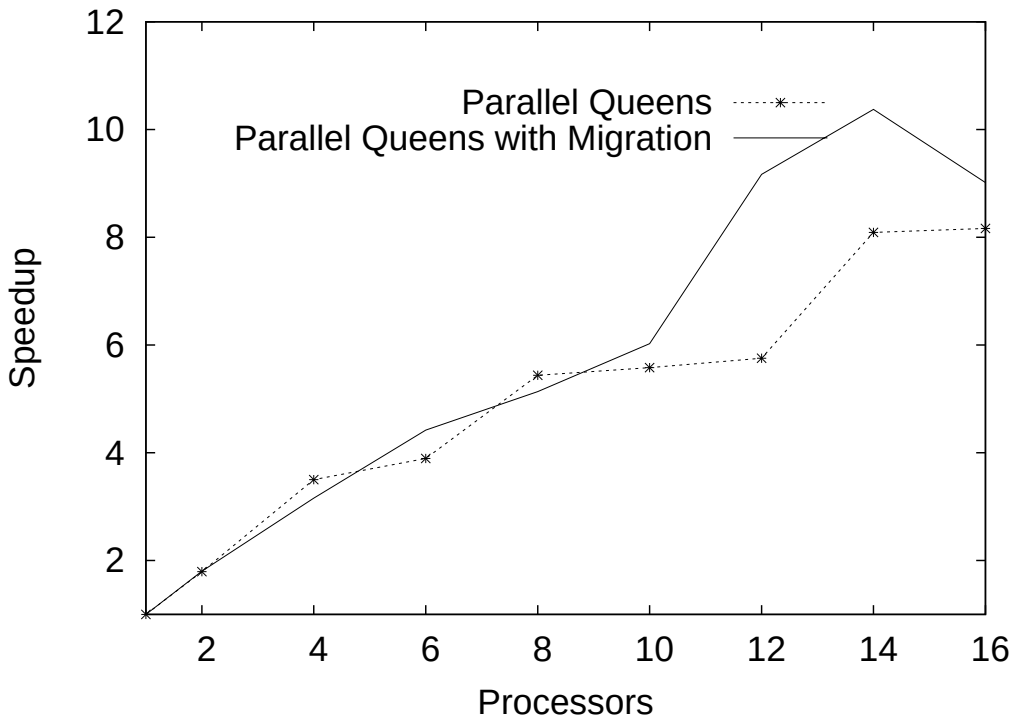


Figure 3.14: Speedups for Queens

that the scheduler chooses migration only if there is no other work available, possibly even accepting short periods of idle time, if fetch requests have been sent already.

A second factor is that programs with a short runtime allow little time for migration to correct a poor initial load distribution. This can be seen in the relatively high variability of sumEuler on 20 and 22 PEs in table 3.2. However, migration still helps even in these cases: the mean, minimum and maximum runtimes are always smaller than for the program without migration.

3.4.3 Overheads

To investigate the overheads of thread migration two simple programs without migratable threads have been measured: one data-parallel and the other divide-and-conquer. For each program table 3.5 reports the paradigm, the mean runtime with and without migration, the average number of threads migrated in each execution and the percentage change in runtime, all measured on 7 PEs. The results show that the migration mechanism has no significant overhead if there are no migratable threads. Moreover, tables 3.2, 3.3 and 3.4 demonstrate that execution with migration enabled on a single PE does not incur significant additional overheads compared to parallel execution

Table 3.4: Queens Runtimes(s) with/without Migration

	Mean Runtime		Min. Runtime		Max. Runtime		Avg No Thr Mig	% Range		Performance Improvement
No PEs	**Mig.**	No Mig.	**Mig.**	No Mig.	**Mig.**	No Mig.		**Mig.**	No Mig.	Mean Runtime
1	**405.4**	405.6	**405.3**	405.2	**405.7**	405.9	0	**0.0%**	0.1%	0%
2	**225.0**	227.8	**219.6**	227.8	**227.7**	227.9	0	**3.6%**	0.0%	1%
4	**128.4**	116.7	**114.9**	116.6	**169.6**	116.8	1.5	**42.6%**	0.1%	-10%
6	**91.7**	104.9	**78.03**	103.1	**117.6**	108.7	4.8	**43.1%**	5.3%	12%
8	**78.9**	75.1	**72.2**	74.9	**102.1**	75.3	2.5	**37.8%**	0.5%	-5%
10	**67.3**	73.2	**63.2**	67.1	**68.8**	103.0	1.33	**8.3%**	49%	8%
12	**44.2**	70.9	**38.8**	62.9	**70.8**	73.1	1.33	**72.3%**	14.3%	37%
14	**39.0**	50.5	**39.0**	38.9	**39.1**	78.9	0.84	**0.2%**	79.2%	22%
16	**44.9**	50.0	**38.9**	39.0	**73.9**	72.3	1.6	**77.9%**	66.6%	10%

Table 3.5: Migration Overheads

Program Name	Paradigm	Mean Runtime		Avg No Thr Mig	% Change in Runtime
		With Migration	Without Mig.		
ParFib 35	Div & Conq.	**5.6**	5.7	0	+2%
ParMap	Data Par.	**25.2**	25.3	0	+1%

without migration.

3.5 Related Work

Thread migration suffered from bad press due to early news of excessive overhead, and comparable systems that provide light-weight, and typically fine-grained, threads tend to avoid migration [BHK+94, CGSv93, Kes96, KLB91].

In the context of earlier versions of parallel graph reducers, early experiments on the GRIP system, implemented on a special-purpose distributed memory machine, indicated, that despite the availability of several sparking strategies to control and balance parallelism, thread migration is still needed for some applications to guarantee high utilisation [HP92].

The Cid system [Nik95] extends C with primitives for creating ("forking") new threads and synchronising ("joining") them on shared variables, managed in a virtual shared heap. The Cid systems holds runnable threads in two different queues: one for the threads that must be executed on the current processor, one for threads that might

be migrated to another processor. The cost for handling the messages is reduced by using the technique of active messages [vCGS92] where the message carries a pointer to a function, the "handler", that is called when receiving the message. The effectiveness of Cid's load balancing and latency tolerance mechanisms is assessed in [NS94].

Cilk [BJL+95] is similar to Cid, in that it extends C with constructs for creating and synchronising light-weight threads. Its processors also use a sophisticated work-stealing scheduler to obtain new parallelism. While Cilk provides thread migration, it is optimised for local execution of a thread, since that is the most common case. Thus, still high overhead is associated with migration.

The Filaments system [LFA96] is in many aspects similar to our system: it emphasises fine-grained parallelism on a distributed shared heap, with dynamic and implicit management of work and data; the programmer is only required to expose parallelism. It is implemented as an extension of C. Two levels of threads can be distinguished: filaments, with a code pointer and arguments but without a stack, and server threads, with an attached stack, acting as a scheduler over a set of filaments. In balancing the load of the system, it employs a sophisticated adaptive data placement mechanism, that tries to minimise access to remote data and uses information gained from dynamic monitoring of data access. Overall, the system focuses on the placement of data rather than threads.

The Ariadne Threads system [MR96] is C-based and implements user-level threads and explicit thread migration on shared and distributed memory machines. In contrast to Filaments, placement decisions focus on threads, rather than data, by migrating a thread to the location of its data, rather than vice versa.

The Amber system [CAL+89] uses a virtual shared memory model, implemented on the Topaz operating system and is programmed in C++. It provides library calls to realise dynamic clustering of threads at runtime, via explicit thread migration. In contrast to Ariadne it puts limitations on the total number of threads per node, but gains reduced packing and thread management overhead.

Another virtual shared memory system focusing on thread migration is Millipede [ISS98]. It uses kernel-threads and is very flexible, allowing explicit migration at almost any point in the execution. It uses dynamic mechanisms migrating both threads and data to maximise data locality. Stack packing is simplified by guaranteeing that stacks will occupy the same place on all processors. This reduces packing costs but wastes some

memory space.

As shown in Chapter 2, a lot of research was done on process migration in the area of operating systems in the late 70s [MDW99]. The objective of introducing process migration into an operating system was to improve load balancing and fault tolerance. Thus, process migration should be transparent to the user and is not under the control of the programmer. Many operating systems were designed to support process migration, some well known examples are Mach [BRS+85] and MOSIX [BL98].

We are primarily interested in thread migration as a way to obtain even load balance and thereby improve performance in a system for parallel computation. Alternative applications of this technique, with different design requirements, are the use of migration in persistent systems [MMS95] and for mobile computation on open networks, as presented in the next Chapters.

3.6 Summary

In this Chapter we have described a system for implicit mobility of computations in a purely functional language. We have presented a design and implementation of thread migration for the GUM runtime system underlying GpH. The design exploits the uniform representation of heap objects in the STG-machine: the packing of a TSO and its stack only requires an extension of the default packing mechanism to handle a new kind of heap closure. The design also makes minimal extensions to the relatively simple GUM communication protocol, adding only two new kinds of messages. Here we profit from both data and threads being represented by heap objects.

Performance measurements of five programs on a high-latency Beowulf cluster show that thread migration in GUM can improve the performance, and reduce the variability in performance, of data-parallel and divide-and-conquer programs with low processor utilisation. In summary: sumEuler is 35% faster on 22 PEs, Maze is 21% faster on 16 PEs, and Queens is 10% faster on 16 PEs. Measurements of these and two other programs show that migration does not incur significant overheads if there are no migratable threads, or on a single PE. However, it is hard for migration to improve programs that already have good utilisation, and migration may both increase variability and occasionally reduce performance. For example neither Maze nor Queens is significantly improved by migration until 10 or more PEs are used.

Many of the tools designed for implicit mobility could also be used in an explicit mobile language. A graph packing mechanism to communicate computations is also needed in an explicit mobile language, and a very similar packing algorithm is used in *m*Haskell, the explicit mobile language that is described in Chapter 5. The implementation techniques for thread migration presented in this Chapter could also be used to add explicit strong mobility to a distributed functional language. Migration of the complete representation of a thread in the RTS is not the only way of implementing strong mobility, and a different alternative is given in Section 6.3.

Chapter 4

Monads and Stateful Computations in Haskell

This Chapter presents the basic ideas and intuitions behind monads, monadic IO and concurrency in Haskell. Programs written in an explicit mobile language are usually stateful computations that interact with resources at the locations that they visit. These stateful computations must be carefully managed in a purely functional language to preserve the purity of the language. In the purely functional language Haskell, such computations are kept in an `IO` monadic data type that separates the purely functional code from the stateful side-effecting computations. First we review the concept of a monad and the language constructs used to operate on monadic values in Haskell. Next we present the IO monad and describe how stateful computation are used in Haskell. In Section 4.2.1, Concurrent Haskell is described. Finally, in Section 4.3 the semantics for monadic I/O and Concurrent Haskell is reviewed. A good tutorial on Monadic I/O and Concurrent Haskell is given in [PJ01].

4.1 Monads

The concept of monad comes from a branch of mathematics called category theory. Philip Wadler [Wad90, Wad95], inspired by Moggi's work on denotational semantics and monads [Mog89], proposed monads as a general technique for structuring functional programs. Monads can also be used to model impure features in a purely functional language, such as exception, state, continuations and concurrency. A *monad* is a triple

consisting of a type constructor `m` and two polymorphic functions:

```
return :: a -> m a
(>>)   :: m a -> (a -> m b) -> m b
```

$$\texttt{return x} >>= \texttt{f} \;=\; \texttt{f x} \qquad (LUNIT)$$

$$m >>= \texttt{return} \;=\; m \qquad (RUNIT)$$

$$\frac{x \notin fn\,(m_3)}{m_1 >>= (\lambda x.m_2 >>= (\lambda y.m_3)) = (m_1 >>= (\lambda x.m_2)) >>= (\lambda y.m3)} \qquad (BIND)$$

Figure 4.15: Monad Laws

These functions must satisfy the laws in Figure 4.15. The `return` function is used to insert a value into a monad and the *bind* function `>>=` implements sequential composition. A program may use many different monads at the same time and the type class mechanism in Haskell allows the overloading of the *bind* and `return` operations for the different types. The `Monad` class can be seen in figure 4.16.

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Figure 4.16: The `Monad` class

As a simple example, consider the definition of the `Maybe` monad:

```
data Maybe a = Nothing | Just a


instance Monad Maybe where
    return         = Just
    fail           = Nothing
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
```

The `Maybe` monad is used to combine computations that may return `Nothing`. It is similar to the idea of exceptions in programming languages: a computation may return a value (`Just a`), or fail returning `Nothing`. If the `>>=` operation receives a `Just a`

value, it continues the computation by applying its function argument to `a`, otherwise it returns `Nothing`.

The following example uses the `Maybe` monad to prototype functions to access a Database system:

```
connectDB :: DBName -> Maybe Con

query :: String -> Con -> Maybe String

processQuery :: String -> Maybe String

useDB :: Maybe String
useDB = connectDB "myDB" >>=
        \con -> query "select * from table" con >>=
        \res -> processQuery res
```

The `Maybe` monad used in the example helps dealing with computations that may fail. If one of the functions returns `Nothing` the `Maybe` monad stops the flow of computations returning `Nothing` as the result of the whole program.

Haskell offers a special syntax for monadic programming called "the `do` notation", that makes the functional code look similar to blocks of code in an imperative language. A new version of `useDB` using the `do` notation would be:

```
useDB2 :: Maybe String
useDB2 = do c <- connectDB "myDB"
            r <- query "select * from table" c
            processQuery r
```

For bigger programs is much easier to use the `do` syntax instead of monadic composition and lambda abstractions. The `do` notation is just syntactic sugar, it is translated by the compiler into calls to `>>=` as can be seen in Figure 4.17. In the translation is possible to see the expression `(x <- e)` simply *binds the variable* `x`, which is different than an assignment in an imperative language.

```
do { x <- e ; s}  =  e >>= \x -> do { s }
do { e ; s}       =  e >> do { s }
do { e }          =  e
```

Figure 4.17: The `do` notation

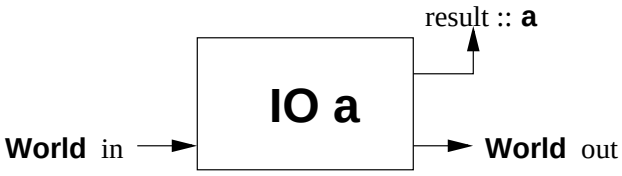


Figure 4.18: The IO data type

4.2 Monadic IO

Computations with side-effects were always a problem in purely functional languages with lazy evaluation. Functional languages are based on pure functions that have no side-effects. Furthermore, in a call-by-need language there is no specific order of evaluation for expressions, and it is difficult to predict when side effects (e.g., printing) will be executed during the evaluation of expressions, if they are executed at all. In Haskell, stateful computations are encapsulated inside the IO monadic data type, that separates pure functions from stateful functions, and imposes an order of evaluation for expressions of type IO.

A value of type `IO a` is an *action* that, when performed, may do some input/output, before delivering a value of type `a` [PJ01]. Using the `IO` data type, a function that prints a character on the screen would have the following type:

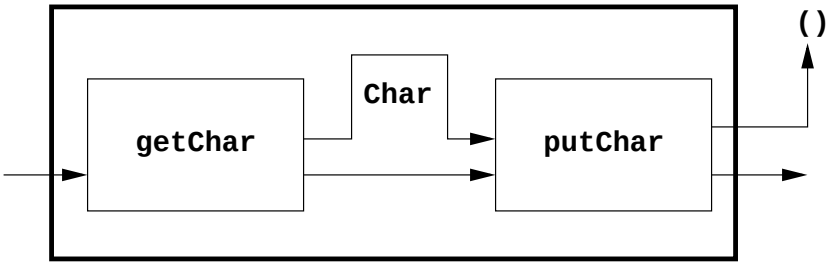
```
putChar ::  Char -> IO ()
```

The `putChar` function takes a character as an argument and returns an IO action that when performed, will print the character on the screen and return the value `()`. Usually it helps to visualise an IO action like `IO a` as a function that takes as an argument the current state of the `World`, and returns a new `World` and a value of type `a`, as can be seen in Figure 4.18. Viewing a stateful computation as a *state transformer* helps reasoning about programs with side-effects and preserves the purity of the language [LPJ95].

The IO monad provides two primitives that implement the basic polymorphic functions of a monad (`>>=` and `return`, as in Figure 4.19).

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

Figure 4.19: Basic IO operations



getChar >>= putChar

Figure 4.20: The >>= operator

Using >>=, if we have an action

```
getChar :: IO Char
```

that when performed reads a character from standard input, it is possible to implement an echo function:

```
echo :: IO ()
echo = getChar >>= (\c -> putChar c)
```

The bind combinator performs the action of its first argument and passes the result to its second action (as can be seen in Figure 4.20). It is useful to have yet another operation (>>) that is used when the programmer wants to execute two actions in sequence, ignoring the result of the first before executing the second. The (>>) function can be implemented in terms of bind:

```
(>>) :: IO a -> IO b -> IO b
(>>) a b = a >>= (\x -> b)
```

Using (>>)) we can write:

```
echo2 :: IO ()
echo2 = echo >> echo
```

The third operation is `return` that injects a value into a monad. In the case of the `IO` type, it does no I/O and returns its argument without any side-effect. Using `return` is possible to implement IO actions that return results, e.g., the `getLine` function:

```
getLine :: IO [Char]
getLine = do
           c <- getChar
           if c=='\n'
             then return []
             else do
                   cs <- getLine
                   return (c : cs)
```

Once a value is inside the IO type there is no way of taking it out, which guarantees that the properties of the purely functional language will be preserved and that side-effects will occur only in the right context.

Another advantage of using monads in a functional language is that new control structures can be implemented that encapsulate patterns of computation. For example, `sequence`, executes a list of Monadic values:

```
sequence :: Monad m => [m a] -> m [a]
sequence []  =  return []
sequence (x:xs) = do r  <- x
                     rs <- sequence xs
                     return (r:rs)
```

Sometimes, the programmer may want to execute IO actions only for their effect:

```
sequence_ :: Monad m => [m a] -> m ()
sequence  = foldr (>>) (return ())
```

The function `sequence_` executes the actions and throws away their results. Using the functions `sequence` and `sequence_`, it is possible to write versions of the `map` function that work over a list of Monadic values:

```
mapM :: Monad a => (b -> a c) -> [b] -> a [c]
```

```
mapM f as =  sequence (map f as)
```

```
mapM_ :: Monad a => (b -> a c) -> [b] -> a ()
mapM_ f as =  sequence_ (map f as)
```

The function `mapM` applies a function that generates a monadic value to all the elements of a list, executing the resulting actions. `mapM_` executes the actions only for their effect, hence it does not return any results.

4.2.1 Concurrent Haskell

Concurrent Haskell [PJGF96] is an extension to Haskell 98 to support concurrent programming.

The main primitive for concurrency is `forkIO` (Figure 4.21), it takes an I/O action as an argument and forks a new thread that runs concurrently with the parent thread, i.e., the one that made the call to `forkIO`. The `forkIO` primitive returns immediately, returning as a result an identifier for the forked thread. A thread can be put to sleep for a number of microseconds by calling `threadDelay`.

```
forkIO       :: IO a -> IO ThreadId
threadDelay  :: Int -> IO ()
killThread   :: ThreadId -> IO ()
```

Figure 4.21: Concurrency primitives

In Figure 4.22 is an example of a Concurrent Haskell program. It forks two threads, one that prints the string `"Hello "` while the other thread prints `"World"`. There is no way of telling how the strings will be printed on the screen as the execution may be interleaved.

If we want threads to cooperate, a synchronisation mechanism is needed. In Concurrent Haskell, synchronisation is obtained by using MVars (see Figure 4.23). An `MVar` is a mutable location, providing implicit synchronisation, that can be either empty or contain a value of type `a`. The `putMVar` primitive writes a value into the MVar, and in case it is full, it blocks until the MVar becomes empty. A call to `takeMVar` will empty

```
main =  do
   forkIO th
   forkIO th2
 where
  th  :: IO ()
  th  = putStr "Hello "
  th2 :: IO ()
  th2 = putStr "World"
```

Figure 4.22: Example using threads

the MVar, and in case it was empty it will block until another thread fills the MVar by calling `putMVar`.

```
data MVar a -- abstract

newEmptyMVar  :: IO (MVar a)
takeMVar      :: MVar a -> IO a
putMVar       :: MVar a -> a -> IO ()
isEmptyMVar   :: MVar a -> IO Bool
```

Figure 4.23: MVars

In the example of Figure 4.24, an MVar is used to synchronise two threads, so `th2` will always print the string `"World!!\n"` after the thread `th1` has printed the string `"Hello "`.

Concurrent Haskell is described in details in [PJGF96, PJ01].

4.3 Semantics of Monadic IO and Concurrency

4.3.1 Basic IO

The semantics for monadic IO described in this text is a review of the the work presented in [PJ01, MPJMR01], and is the basis for the semantics for the mobile language described in Chapter 5. The semantics is divided in two levels: an *inner denotational semantics* for the behaviour of pure terms that is not described here, e.g., could be based on [MLPJ99]; and an *outer monadic transition semantics* for the behaviour of `IO` computations.

In figure 4.25 we present the syntax of a simple Haskell like functional language. M

```
main =  do
   mvar <- newEmptyMVar
   forkIO (th1 mvar)
   forkIO (th2 mvar)
 where
  th1 :: MVar () -> IO ()
  th1 mvar = do
      putStr "Hello "
      putMVar mvar ()
  th2 :: MVar () -> IO ()
  th2 mvar = do
      _ <- takeMVar mvar
      putStr "World!!\n"
```

Figure 4.24: Example using threads and MVars

and N range over *terms* and V over values. A value is something that is considered by the inner, purely-functional semantics as evaluated. Primitive monadic `IO` operations

	x, y	$\in$	*Variable*
	k	$\in$	*Constant*
	con	$\in$	*Constructor*
	c	$\in$	*Char*
Values	V	$::=$	$\backslash x -> M \mid k \mid con M_1 \cdots M_n \mid c$ $\mid$ `return` $M \mid M$ `>>=` N $\mid$ `putChar` $c \mid$ `getChar`
Terms	M, N	$::=$	$x \mid V \mid M\ N \mid$ `if` M `then` N_1 `else` $N_2 \mid \cdots$
Evaluation contexts	$\mathbb{E}$	$::=$	$[\cdot] \mid \mathbb{E}$ `>>=` M

Figure 4.25: Syntax of a Basic Stateful Functional Language

are treated as values. For example, `putChar 'c'` is a value. No further work can be done on this term in the purely-functional world. Note that some of these monadic `IO` values have arguments that are not arbitrary terms but are themselves values (e.g., c). So `putChar 'A'` is a value but `putChar (ch 65)` is not. `putChar` is a *strict* data constructor. The reason for this choice is that evaluating `putChar`'s argument is something that should be done in the purely-functional world.

The transition from one program state to the next may or may not be *labelled* by

an *event*, α. So we write a transition like this:

$$P \xrightarrow{\alpha} Q$$

The event α represents communication with the external environment; that is, input and output.

In order to make it easier to describe which is the next transition rule to be applied, *evaluation contexts* are used. An evaluation context $\mathbb{E}$ is a term with a hole $[\cdot]$, and its syntax is presented in figure 4.25. The symbol $[\cdot]$ indicates the location of the hole in an expression and $\mathbb{E}[M]$ is used to show that the hole in $\mathbb{E}$ is being filled by the term M. Example:

$$\mathbb{E}_1 = [\cdot] >>= (\backslash \texttt{c} -> \texttt{putChar}\ (\texttt{toUpper}\ \texttt{c}))$$
$$\mathbb{E}_1\,[\texttt{getChar}\ ] \;=\; \texttt{getChar} >>= \;(\backslash \texttt{c} -> \texttt{putChar}\ (\texttt{toUpper}\ \texttt{c}))$$

Using evaluation contexts, the basic transition rules for monadic actions are given in Figure 4.26. The first rule says that a program consisting only of `putChar` c can make a transition, labelled by $!c$, to a program consisting of `return ()`. The second rule describes in a similar way the input of a value. The third rule explains how to evaluate pure functional expressions in a program. If a term M has value V, computed by the denotational semantics of M ($\varepsilon[[M]]$), then M can be replaced by V at the active context.

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

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

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

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

Figure 4.26: Basic Transition Rules

In Figure 4.27 we present the evaluation of the following program:

```
main = getChar >>= \c ->
       putChar (toUpper c)
```

using the rules of Figure 4.26. The first transition is made by using the context $\mathbb{E}_1$ (presented before) and the rule $(GETC)$. In the example we assume that character delivered by `getChar` was `'a'`. The next rule that can be applied is $(LUNIT)$, and the result is a function application. This evaluation is carried out by the denotational semantics of the purely-functional part of the language as stated in rule (FUN), using normal beta-reduction. Notice that as `putChar` is a strict constructor, $\varepsilon[[]]$ produces `putchar 'A'` and not `putChar (toUpper 'a')`. Next, the evaluation continues by applying the rule for `putChar` that prints the character and then the program is finished.

$$
\begin{array}{lll}
 & \{\,\texttt{getChar} >\!\!>\!= \ (\backslash \texttt{c} -\!> \texttt{putChar}\ (\texttt{toUpper}\ \texttt{c}))\} & \\
\xrightarrow{?\text{`a`}} & \{\,\texttt{return `a`} >\!\!>\!= \ (\backslash \texttt{c} -\!> \texttt{putChar}\ (\texttt{toUpper}\ \texttt{c}))\} & (GETC) \\
\rightarrow & \{(\backslash \texttt{c} -\!> \texttt{putChar}\ (\texttt{toUpper}\ \texttt{c}))\ \texttt{`a`}\} & (LUNIT) \\
\rightarrow & \{\texttt{putChar `A`}\} & (FUN) \\
\xrightarrow{!\text{`a`}} & \{\,\texttt{return}\ ()\} & (PUTC)
\end{array}
$$

Figure 4.27: Example of evaluation using transition rules

4.3.2 Concurrency and MVars

In figure 4.28, the syntax presented in the last section is extended to support MVars and concurrency. New values are added to represent the new IO operations, the name of an MVar m, a thread t and a thread delay d. The program states are extended by adding MVars and named threads where $\langle M \rangle_m$ represents a MVar m containing value M, $\langle\rangle_m$ represents the empty MVar, and $\{M\}_t$ represents a thread called t. The program state can be the parallel composition of threads and MVars. The structural congruence and structural transitions for program states are presented in figure 4.30.

The new transition rules are given in figure 4.29. The rules $(FORK)$ and $(NEWM)$ work in a similar way: if the next IO action in the program is to create a new MVar or thread, it makes a transition to a new state in which the main program is in parallel with the new MVar/thread. The names u and m are arbitrary names that must not already be used in M or in the evaluation context $\mathbb{E}$.

The $(TAKEM)$ rule says that if the next IO action to be performed is `takeMVar` m, and it runs in parallel with a MVar m containing the value M, it can be replaced by `return` M and the MVar is emptied.

$$
\begin{array}{lcl}
m & \in & \texttt{Mvar} \\
t & \in & \texttt{ThreadId} \\
d & \in & Integer \\
 & & \\
V & ::= & ...\texttt{forkIO}\ M \mid \texttt{threadDelay}\ d \mid t \mid d \\
 & \mid & \texttt{putMVar}\ m\ N \mid \texttt{takeMVar}\ m \mid \texttt{newEmptyMVar} \mid m \\
 & & \\
P, Q, R & ::= & ... \\
 & \mid & \{M\}_t \quad \text{A thread called } t \\
 & \mid & P|Q \quad \text{Parallel Composition} \\
 & \mid & \langle M \rangle_m \quad \text{An } \texttt{Mvar} \text{ called } m \text{ containing } M \\
 & \mid & \langle\rangle_m \quad \text{An empty } \texttt{MVar} \text{ called } m \\
 & \mid & \nu x.P \quad \text{Restriction}
\end{array}
$$

Figure 4.28: Extended Syntax for Concurrency

$$
\frac{u \notin fn\,(M, \mathbb{E})}{\{\mathbb{E}[\texttt{forkIO}\ M]\}_t \rightarrow \nu u.(\{\mathbb{E}[\texttt{return}\ u]\}_t \mid \{M\}_u)} \quad (FORK)
$$

$$
\frac{m \notin fn\,(\mathbb{E})}{\{\mathbb{E}[\texttt{newEmptyMVar}]\}_t \rightarrow \nu m.(\{\mathbb{E}[\texttt{return}\ m]\}_t \mid \langle\rangle_m)} \quad (NEWM)
$$

$$
\{\mathbb{E}[\texttt{takeMVar}\ m]\}_t \mid \langle M \rangle_m \rightarrow \{\mathbb{E}[\texttt{return}\ M]\}_t \mid \langle\rangle_m \quad (TAKEM)
$$

$$
\{\mathbb{E}[\texttt{putMVar}\ m\ M]\}_t \mid \langle\rangle_m \rightarrow \{\mathbb{E}[\texttt{return}\ ()]\}_t \mid \langle M \rangle_m \quad (PUTM)
$$

$$
\{\{\mathbb{E}[\texttt{threadDelay}\ d]\}_t\} \xrightarrow{\$d} \{\{\mathbb{E}[\texttt{return}\ ()]\}_t\} \quad (DELAY)
$$

Figure 4.29: Extended Transition Rules for Concurrency and MVars

Note that now the semantics has became non-deterministic: if two threads take a value out of a MVar, the semantics does not specify which will succeed. Once the MVar is empty the other thread will not be able to continue while another value is not put into the MVar.

The *blocking* of a thread is not modelled in the rules. If a thread tries to get a value out of an empty MVar, what happens is that there is no valid transition rule to be applied, hence the thread stays blocked until another thread puts a value into the MVar.

In the transition for `threadDelay`, a new event ($\$d$) is introduced, which means that *d microseconds elapsed*. In this case, delay is modelled as an interaction with an external clock.

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & (COMM) \\
P \mid (Q \mid R) & \equiv & (P \mid Q)\ R & (ASSOC) \\
P \mid (Q \mid R) & \equiv & (P \mid Q)\ R & (ASSOC) \\
\nu x.\nu y.P & \equiv & \nu y.\nu x.P & (SWAP) \\
(\nu x.P) \mid Q & \equiv & \nu x.(P \mid Q), \quad x \notin fn(Q) & (EXTRUDE) \\
\nu x.P & \equiv & \nu y.P[y/x], \quad y \notin fn(P) & (ALPHA)
\end{array}
$$

$$
\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \quad (PAR)
$$

$$
\frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q} \quad (NU)
$$

$$
\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \quad (EQUIV)
$$

Figure 4.30: Structural congruence, and structural transitions

Here is an example program that will be reduced using the semantic rules:

```
main = newMVar 0 >>= \m ->
       takeMVar m >>= \v ->
       putMVar m (v+1)
```

The program creates a new MVar using **newMVar**, that has a similar transition rule to (*NEWM*), the difference being that it creates a MVar filled with an initial value. Then it takes the value, increments it, and writes it back into the MVar.

In Figure 4.31, the program is reduced using the transition rules (names have been abbreviated to save space).

$$
\begin{array}{rll}
 & \{\texttt{newM 0} >\!\!>= \backslash\texttt{m} -\!\!> \texttt{takeM m} >\!\!>= \backslash\texttt{v} -\!\!> \texttt{putM m (v + 1)}\}_t & \\
\rightarrow & \nu m.(\{\texttt{return } m >\!\!>= \backslash\texttt{m} -\!\!> \texttt{takeM m} >\!\!>= \backslash\texttt{v} -\!\!> \texttt{putM m (v + 1)}\}_t \mid \langle 0\rangle_m) & (NEWM) \\
\rightarrow & \nu m.(\{(\backslash\texttt{m} -\!\!> \texttt{takeM m} >\!\!>= \backslash\texttt{v} -\!\!> \texttt{putM m (v + 1)})\ m\}_t \mid \langle 0\rangle_m) & (LUNIT) \\
\rightarrow & \nu m.(\{\texttt{takeM } m >\!\!>= \backslash\texttt{v} -\!\!> \texttt{putM } m \texttt{ (v + 1) }\}_t \mid \langle 0\rangle_m) & (FUN) \\
\rightarrow & \nu m.(\{\texttt{return 0} >\!\!>= \backslash\texttt{v} -\!\!> \texttt{putM } m \texttt{ (v + 1) }\}_t \mid \langle\rangle_m) & (TAKEM) \\
\rightarrow & \nu m.(\{(\backslash\texttt{v} -\!\!> \texttt{putM } m \texttt{ (v + 1)) 0 }\}_t \mid \langle\rangle_m) & (LUNIT) \\
\rightarrow & \nu m.(\{\texttt{putM } m \texttt{ (0 + 1) }\}_t \mid \langle\rangle_m) & (FUN) \\
\rightarrow & \nu m.(\{\texttt{return ()}\}_t \mid \langle 0+1\rangle_m) & (PUTM)
\end{array}
$$

Figure 4.31: Example of evaluation using the rules for MVars

4.4 Summary

Monads are a general technique to structure functional programs. A monad describes how to combine computations into a new computation and defines functions that abstract away the code for combining computations. There are a number of different monads and this Chapter described how to perform I/O and Concurrency in Haskell using the `IO` Monad. The `IO` monad solves two of the main problems in performing stateful computations in a lazy purely functional language: order of evaluation and side effects. In a lazy language, there is no order of evaluation for expressions and it is difficult to predict when a command, e.g., for printing a value on the screen, will be executed. The `IO` monad solves this problem by introducing an order for the evaluation of impure expressions. The `IO` monad also helps with the problem of introducing effects in a purely functional language, as it makes the effects explicit in the type system. Hence in Haskell, the `IO` monad works as a sub-language for writing stateful programs, and the purely functional part of the language can be used to write new control structures that manipulate stateful computations. Although monads can be written in any programming language, Haskell provides special support for them through its type classes.

Chapter 5

Mobile Haskell

The high-level techniques used in process migration or *implicit mobility* does not scale well on large scale distributed systems, such as the Internet [MDW99, FPV98].

In mobile languages that work on global networks, the underlying network must be visible to the program, hence programming is location aware and the mobility of computations is under the programmer's control i.e., mobility is *explicit*, as discussed in Chapter 2.

Although mobile computation provides a more scalable programming paradigm for global networks compared to traditional approaches, distributed mobile systems are difficult to engineer, manage and reason about as programs do not remain static in one location. We argue that a language for mobile computation should be based on a *very small set of primitives*, with a *well understood semantics*, that should act as *building blocks for higher-level abstractions* in the language. The primitives should be sufficiently low level to be *efficiently implemented*, have simple clean semantics, and provide *abstractions similar to process calculi*. This Chapter presents *m*Haskell (*Mobile Haskell*), a superset of Concurrent Haskell (see Figure 5.32), for the implementation of distributed mobile programs. *m*Haskell extends Haskell with a small set of higher-order communication primitives including MChannels or *Mobile Channels*. MChannels allow the communication of any Haskell value, including IO computations, functions, and channel names.

The objective of an *explicit* mobile program is usually to exploit the resources available at specific locations. Resources include databases, programs, or specific hardware. As a result mobile programs are usually stateful and in a purely functional language

must be carefully managed to preserve referential transparency. This is in contrast to parallel functional languages, where stateless computations are freely distributed across locations to reduce runtime. Stateful operations in Haskell, as described in Chapter 4, are encapsulated inside of an abstract data type of I/O actions, and in *m*Haskell, all mobility primitives are kept inside of the IO monad. Haskell computations are first-class values, i.e., functions can receive actions as arguments, return actions as results, and actions can be combined to generate new actions. The crucial implication for a mobile language is that computations can be manipulated and new abstractions over computations defined.

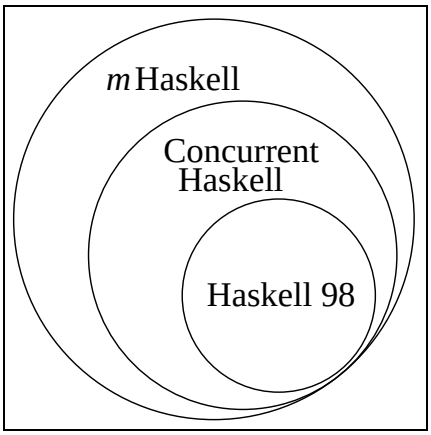


Figure 5.32: *m*Haskell is an extension of Concurrent Haskell

5.1 *m*Haskell Design

5.1.1 MChannels

Mobile Haskell or *m*Haskell is a modest conservative extension of Concurrent Haskell [PJGF96]. It enables the construction of mobile applications by introducing higher order, single reader, communication channels called *Mobile Channels*, or MChannels. MChannels allow the communication of arbitrary Haskell values including functions, IO actions and channels. Figure 5.33 shows the MChannel primitives.

The operational semantics of these primitives is given in Section 5.2, but we start with an informal explanation. The `newMChannel` function is used to create a mobile channel and the functions `writeMChannel` and `readMChannel` are used to write/read data from/to a channel. MChannels provide synchronous communication between

```
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 5.33: Mobile Channels

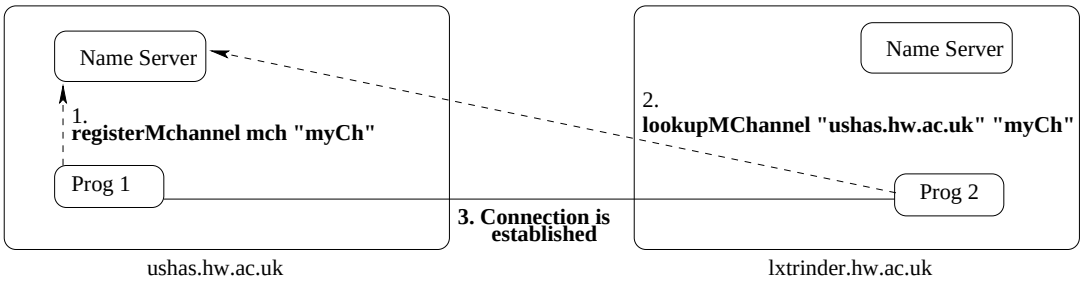


Figure 5.34: Example using MChannels

hosts. When used locally, they 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 machine of the system and a channel is always registered in the local name server with the `registerMChannel` function. MChannels are single-reader channels, meaning that only the program that created the MChannel can read values from it. Values are evaluated to normal form before being communicated, as explained in Section 5.1.4.

Figure 5.34 depicts a pair of simple programs using MChannels, and the code for this program is in Figure 5.35. First a program running on a machine called `ushas` registers a channel `mch` with the name `"myCh"` in its local name server. When registered,

```
--code on ushas:

main = do
  mch <- newMChannel
  registerMChannel mch "myCh"
  result <- readMChannel mch
  result

-- code on lxtrinder:

main = do
   mch <- lookupMChannel "ushas.hw.ac.uk" "myCh"
   case mch of
     Just channel  -> writeMChannel channel (print "Hello!")
     Nothing       -> print ("Error: Couldn't find channel!")
```

Figure 5.35: Example using MChannels

the channel can be seen by other locations in the network using the `lookupMChannel` primitive. After the lookup, the connection between the two machines is established and communication is performed with the functions `writeMChannel` and `readMChannel`.

5.1.2 Discovering Resources

One of the objectives of mobile programming is to better exploit the resources available in a network, as discussed in Chapter 2. 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 e.g., databases, local functions, load information, etc.

```
type ResName = String

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

Figure 5.36: Primitives for resource discovery

Figure 5.36 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 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 [Mar05] 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` must be cast into a monomorphic type in order to become a dynamic value.

5.1.3 A Simple Example

Figure 5.37 shows an *m*Haskell program that computes the load of a network. It visits a `listomachines` and executes the computation called `mobile` on all the machines of the list.

The function `sendMobile` is mapped over the `listofmachines` by the `mapM` on the second line of the program. It creates a new channel used to get the result of the computation back to the main program, and sends `mobile` to be executed remotely, through the `"servermch"` channel. The locally defined function `exec` simply executes the computation and sends the result back through the `mch` channel.

The server, that must execute on every location in `listofmachines`, can be seen in Figure 5.38. It registers a channel `mch`, and a resource `getLoad` in its local naming services and enters in a loop that reads actions from a channel and executes them.

```
main = do
  list <- mapM (sendMobile mobile) listofmachines
  let v = sum list
  print ("Total Load of the network: " ++ (show v))
  where
   mobile = do
     res <- lookupRes "getLoad"
     case res of
       Just getLoad -> getLoad
       Nothing      -> return 0
   listofmachines = (...)

sendMobile:: IO Int -> HostName -> IO Int
sendMobile comp host = do
  mch <- newMChannel
  mc <- lookupMChannel host "servermch"
  case mc of
    Just cmc -> do
                writeMChannel cmc (exec mch comp)
                readMChannel mch
    Nothing  -> return 0
  where
    exec mch comp = do
        result <- comp
        writeMChannel mch result
```

Figure 5.37: Program that computes the load of a network

This program, as described in Section 5.1, can be written more elegantly using the high level abstractions presented in Chapter 5.

5.1.4 Evaluating Expressions Before Communication

MChannels are hyper-strict: values communicated are evaluated before being sent. The reason for this design decision is that lazy evaluation makes it difficult to reason about what is being communicated. Consider the following example:

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

Suppose that the first element (`a`) of the tuple returned by `f x` is a Boolean, the second (`b`) an integer, and the third (`c`) is a large data structure. Based on the value of `a`, the

```
main = do
        mch <- newMChannel
        registerMChannel mch "servermch"
        registerRes getLoad "getLoad"
        server mch
          where server mch = do
                  action <- readMChannel mch
                  action
                  server mch
```

Figure 5.38: Code for the server

program selects 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. In the beginning of the evaluation, the expression is represented by a graph similar to the one in figure 5.39.

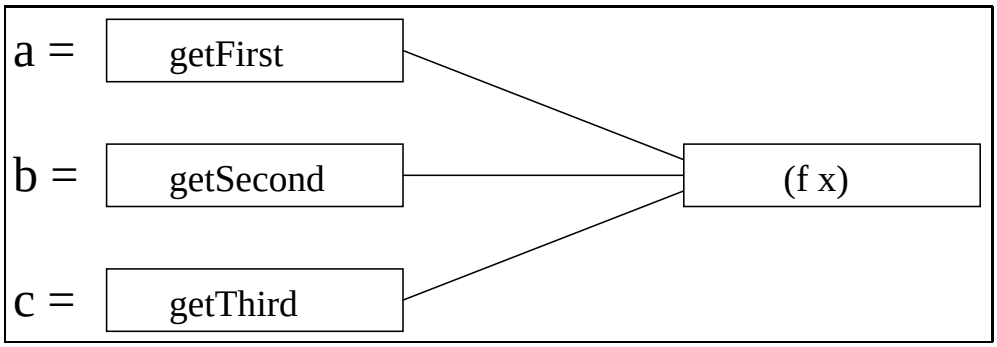


Figure 5.39: Graph for `let (a,b,c) = f x`

At the point where `writeMChannel` is performed, the value `b` is represented in the heap as the *selector* that gets 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 this is not apparent in the Haskell code.

The evaluation of *thunks* (reducible expressions) affects only pure expressions, or expressions that can be evaluated using `seq`. IO computations will not be executed during this evaluation step as they are already in normal form, as can be seen in the semantics of IO actions in Section 4.3.

At a first glance, this evaluation step may seem "unnatural" in a language like Haskell. But in reality, it does not really affect most programs. The objective of a mobile program is usually to interact with resources on remote locations. Hence,

mobile programs are usually stateful computations in the IO monad, and, as explained before, the evaluation step does not execute IO actions.

There are still ways of sending pure expressions to be evaluated on remote hosts, e.g., a tuple with a function and its arguments can be sent, and the function is applied to the values only on the remote end.

5.1.5 Sharing Properties

Many non-strict functional languages are implemented using graph reduction, where a program is represented as a graph and the evaluation of the program is performed by rewriting the graph. The graph ensures that shared expressions are evaluated at most once [PJ92b].

Maintaining sharing between nodes in a distributed system would result in a generally large number of extra-messages and call-backs to the machines involved in the computation (to request structures that were being evaluated somewhere else or to update these structures). In 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 laptop). If we preserve sharing, it 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 already partially solved by making the primitives strict: expressions will be evaluated just once and only the result is communicated.

In *m*Haskell, computations are *copied* between machines and no sharing is preserved *across* machines, although sharing is preserved in the value being communicated.

5.1.6 Single-Reader Channels

MChannels are single-reader channels, for two main reasons. First, it is difficult to decide where a message should be sent when we have more than one machine reading values from the same channel. The main question is where is this channel located? Channels with multiple readers need to maintain a distributed state, keeping track of all the machines that have references to the channel, and these references must be updated every time the channel is moved to another place.

A simple way to implement multiple reader channels is to use a *home server*, keeping the channel in one place, the place where it was created, and all other references to the channel read and write values into the channel by sending messages to this main location. The problem with this approach is that if the main location crashes all the other locations that have references to the channel cannot communicate anymore (Figure 5.40). An implementation of multiple-reader channels using a home server and single-reader MChannels, is described in Section 5.4.1.

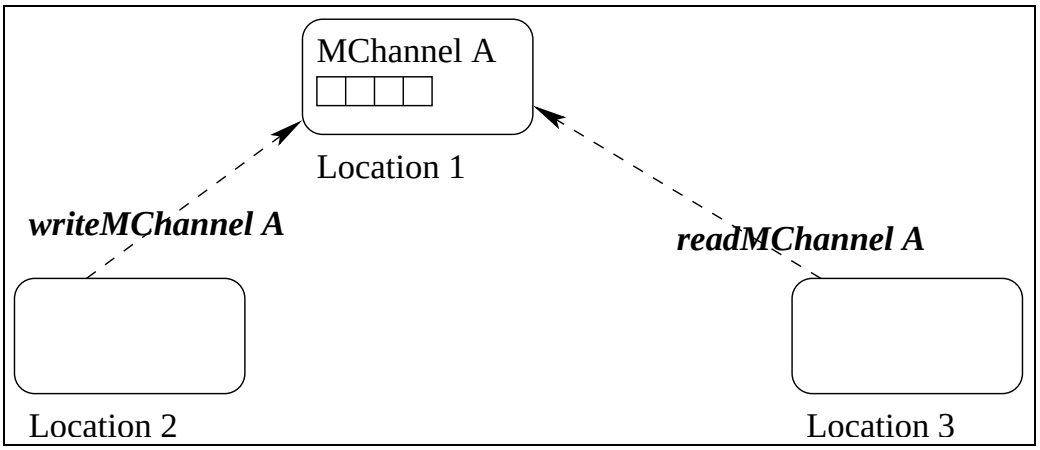


Figure 5.40: Machines 2 and 3 cannot communicate if Machine 1 crashes

The second reason is security: with multiple reader channels one process can *pretend* to be a server and steal messages. This is a classic problem also found in the untyped π-calculus [Mil99], as illustrated in Section 5.4.1.

5.2 Operational Semantics

This Section gives the operational semantics for MChannels by extending the semantics for monadic IO presented in [PJ01, MPJMR01], and reviewed in Chapter 4. The semantics is relatively low-level, and accurately models the implementation of MChannels. As described in Chapter 4, the semantics has two levels: an *inner denotational semantics* for pure terms that is standard and not described here, and an *outer transitional semantics* describing the IO actions and MChannels. Our extensions only affect the transitional semantics.

In Figure 5.41, we extend the syntactic and semantic domains for IO actions with MChannels. New variables are added to represent MChannel identifiers (c), location names (s), MChannel names (n), and remote references to MChannels (r). The new

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

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

Figure 5.41: Extended syntactic and semantic domains of the language with MChannels

The transition rules for MChannels are presented in Figure 5.42. Common congruence rules, such as commutativity and associativity of |, follow from the observation that the semantics for Concurrent Haskell [PJ01] is a special case of our semantics for just one processor. Rule ($NEWC$) creates a new state with an empty MChannel that has no name (ε), and executes in parallel with the current thread. The ($REGC$) and ($UNREGC$) rules set and unset the name n of a MChannel. The `readMChannel` primitive, reads a term M from a local MChannel. It returns the first element in the structure, and cannot be applied to a remote reference, reflecting the fact that the MChannels are single reader channels: only threads running on the location where the MChannel was created can read from it. The `lookupMChannel` function, returns a

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

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

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

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

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

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

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

Figure 5.42: Transition rules for the language with MChannels

remote reference $r_{n@s'}$ to a remote MChannel at location s', if it exists. Otherwise it should return the value `Nothing`, but this is left out of the semantics for simplicity. The `writeMChannel` primitive can be applied to the identifier of a MChannel that runs on the same location or to a remote reference. Rule (*WRITECl*) specifies that writing to a local MChannel appends the value, M, as the last element. Rule (*WRITECr*) states that if `writeMChannel` is executed at location s, and is applied to a reference $r_{n@s'}$ to a channel located at s', the term M is sent to location s' and written into the channel. Before M is communicated the function `forceThunks` is applied to it which

- Forces the evaluation of pure expressions (*thunks*) in the graph of the expression M, without executing the monadic actions
- Substitutes every occurrence of a MChannel in the graph by a remote reference.

The function `forceThunks` is not formalised here, although it can be defined in the semantics of a language with an explicit heap with thunks such as [TA03].

In Figure 5.43 we present the evaluation of the following *m*Haskell program using the semantics:

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

```
        io


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

The `server` program creates and registers an MChannel with a name `n`, and waits for computations on that MChannel. The `client` sends the computation `print "Hello " ++ "World"` to be executed on the server. There are two important things to notice in the evaluation. Firstly, the non-determinism of the semantics: as in a real system, if the `client` looks for a channel before the `server` registers it, the program fails at the ($LOOKUPC$) rule. Secondly in the evaluation of thunks, the strings `"Hello "` and `"World"` are concatenated before communication occurs, but IO actions are not evaluated by `forceThunks`, hence evaluation of `print "Hello world"` occurs only on the server s'.

```
     server :
     L(s', {newMC >>= \mch− > regMC mch n>> readMC mch >>= \io− > io}_t)
→    L(s', νc.({return c >>= \mch− > regMC mch n>> readMC mch >>= \io− > io}_t | ⟨⟩^c_ε))
     (NEWC)
→    L(s', νc.({(\mch− > regMC mch n>> readMC mch >>= \io− > io) c}_t | ⟨⟩^c_ε))   (LUNIT)
→    L(s', νc.({regMC c n>> readMC c>>= \io− > io}_t | ⟨⟩^c_ε))   (FUN)
→    L(s', νc.({readMC c>>= \io− > io}_t | ⟨⟩^c_n))   (REGC)

          client :
          L(s, {lookup s' n >>= \mch− > writeMC mch hello}_t)
     →    L(s, νr.({return r_{n@s'} >>= \mch− > writeMC mch hello}_t))   (LOOKUPC)
     →    L(s, νr.({(\mch− > writeMC mch hello) r_{n@s'}}_t))   (LUNIT)
     →    L(s, νr.({writeMC r_{n@s'} hello}_t))   (FUN)

          server :
→    L(s', νc.({readMC c>>= \io− > io}_t | ⟨print "Hello World"⟩^c_n))   (WRITECr)
→    L(s', νc.({return(print "Hello World") >>= \io− > io}_t | ⟨⟩^c_n))   (READC)
→    L(s', νc.({(\io− > io) (print "Hello World")}_t | ⟨⟩^c_n))   (LUNIT)
→    L(s', νc.({(print "Hello World"}_t | ⟨⟩^c_n))   (FUN)
→    L(s', νc.({(return ()}_t | ⟨⟩^c_n))   (PRINT)
```

Figure 5.43: Example of evaluation using the semantics

5.3 The Implementation

5.3.1 Introduction

The main features of *m*Haskell's implementation are:

- *m*Haskell *is portable.* It is implemented as an extension of the GHC (*Glasgow Haskell Compiler*) [Mar05] compiler that has been ported to many different architectures and operating systems. Our extensions are implemented using standard C and TCP/IP sockets, maintaining a high degree of portability.

- *m*Haskell *is designed to run on heterogeneous networks.* Mobile languages designed to work on global distributed systems, such as the Internet, must be able to communicate code between machines of different architectures and operating systems. The usual approach for communicating computations on heterogeneous networks is by compiling programs into architecture-independent byte-code. GHC combines both an optimising compiler and an interactive environment called GHCi, which compiles user defined functions into byte-code, and this technology could be used by *m*Haskell for communicating computations on heterogeneous networks.

- In *m*Haskell it is possible to combine interpreted and compiled code. GHCi is designed for fast compilation and linking. It generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. As the basic modules in GHC are compiled into machine code and are present in every standard installation of the compiler, the routines for communication have to send only the machine independent part of the program and link it to the local definitions of the machine dependent part when the code is received. This gives us the advantage of having much faster code than using only byte-code, and also reduces the amount of data being communicated.

In the next sections we describe in more detail the implementation.

5.3.2 Packing Routines

The graph representing the computation being communicated is packed at the source and unpacked at the destination. The *m*Haskell pack and unpack routines are based on

the GUM [THM+96] system, described in Chapter 3, but are extended to pack GHCi's Byte-Code Objects (BCOs).

Packing, or *serialising*, arbitrary graph structures is not a trivial task and care must be taken to preserve sharing and cycles. As in GpH [THM+96], GdH [PTL00] and Eden [BLOMP97], packing is done breadth-first, closure by closure and when the closure is packed its address is recorded in a temporary table that is checked for each new closure to be packed to preserve sharing and cycles. The packing routine proceeds packing until every reachable graph has been serialised.

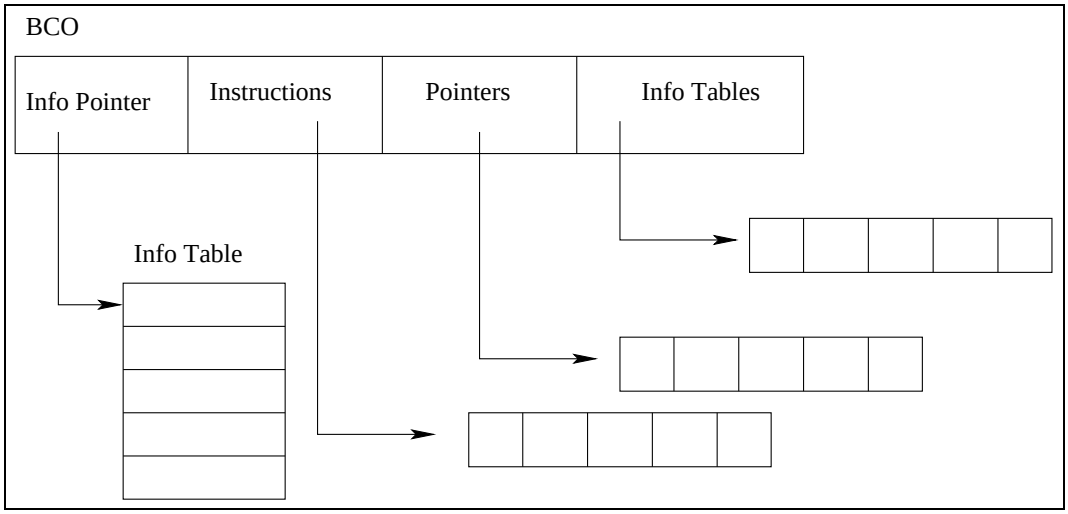


Figure 5.44: The Byte-Code Object

The main heap object to be packed in our implementation of *m*Haskell is the BCO (Figure 5.44), i.e., GHC's internal representation for its architecture-independent byte-code. A BCO is composed of its info_table, which contains information about the closure's fields and also its entry code, a list of instructions, a list of pointers and a list of info tables. The BCO's info table is the same for every BCO and is available in any standard installation of the compiler, so it does not need to be packed. Its list of instructions is just a list of bytes and is packed easily. The list of pointers contains a list of other closures that are used in the byte-code instructions, so all of them must also be packed. The list of info tables contains pointers to info tables of data structures that are constructed during the execution of the BCO's instructions. Those info tables are machine dependent hence are packed in a special way explained in section 5.3.3.

As the basic modules, e.g., prelude, that come with GHC are compiled into machine code and are ubiquitous, i.e., they 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 definition of the machine dependent part when the code is received and unpacked. Once packed, the BCO can be communicated in the way described in section 5.3.5. 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.

Our packing mechanism gives us a simple way of controlling the amount of code communicated: since only functions that are compiled into byte-code are packed, if the programmer knows that one module used in the computation is already in the remote host, this module can be compiled into machine code, so it will not be communicated.

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

We provide a very simple low level interface to the serialisation routines used to implement *m*Haskell. The serialisation primitives are `packV` and `unpackV`:

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

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

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

The example packs, unpacks and executes the function `plusone`. The packing primitives, as detailed in the distributed web server case study presented in Chapter

7, can be used to implement other useful extensions to the compiler, e.g., a library for persistent storage of programs.

5.3.3 Communicating User Defined Types

Currently, user defined data types (ADTs) are always compiled into machine code in GHC, hence the code for the ADTs must be present in every location. There are two ways to overcome this problem. The first one would be to compile the types into a different type of closure that uses BCOs internally. This requires changing the compiler. The other solution is to ship the data type including the values in its info table. The entry code for these objects, which is not very complicated, has to be generated again in the destination.

In the current implementation, all data types used in mobile programs must be defined in all the machines that are going to receive the code. Only the name representing its info table in the linker is packed together with the content of its fields. When unpacking, the routine looks for the local definition of the info table by searching for its name in the linker's tables.

We consider an implementation of one of the two solutions described above, as a tuning step in the development of the prototype implementation, aiming to reduce the common software base needed on all machines.

5.3.4 Evaluating Expressions

Evaluating expressions before communication is not as trivial as it seems. A simple way to evaluate thunks would be to use *evaluation strategies*, as described in Chapter 3:

```
(...)
let list = "Hello " ++ "World!"
 in writeMChannel mch list
```

where in the definition of `writeMChannel` we use the `rnf` strategy to evaluate its argument to normal form.

But strategies will not work in all cases. Consider the following example:

```
f:: a -> b -> Int
```

```
let
  a = (...)
   in writeMchannel ch (f a)
```

In this case it is not possible, inside of the definition of `writeMChannel`, to evaluate the expression `a` using strategies as `writeMChannel` receives only a pointer to the whole closure `f a`. One solution to this problem would be to implement a function `kids` with type:

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

That takes a value from the heap, the expression to be evaluated, and returns an array with all the thunks pointed to by this value. Using `kids` it would be possible to write a `deepSeq :: a -> ()` function that recursively applies `seq` to all the thunks pointed by its argument.

Another way to evaluate thunks is to do it inside the RTS: using a primitive function that creates a new RTS thread to evaluate its argument to normal form by forcing the evaluation of all the expressions pointed by the argument.

*m*Haskell uses an hybrid approach: a thunk in the top level of the graph representing the computation is forced by a `seq` (as in figure 5.45). If there are other thunks in the graph, these thunks are evaluated by an extra thread in the RTS. Care must be taken to preserve the queue of closures yet to be packed if the new thread induces garbage collection. The solution to this problem is to make the packing queue visible to the Garbage Collector.

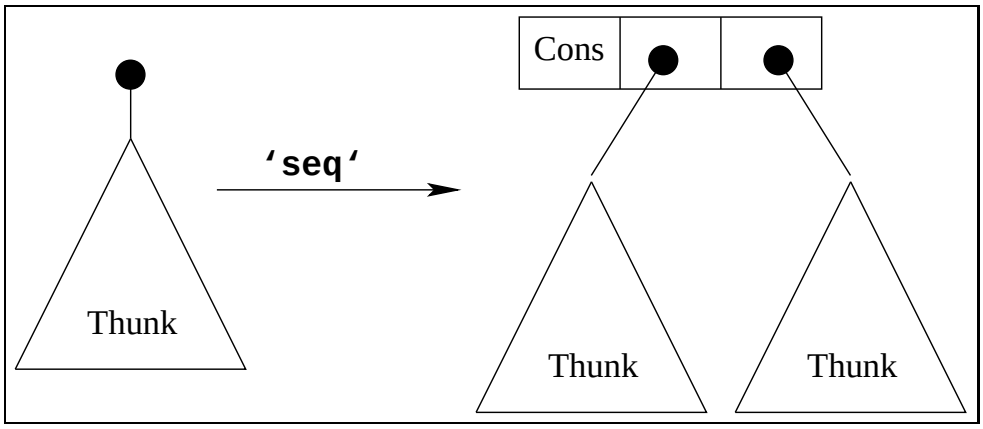


Figure 5.45: Evaluation of thunks using `seq`

5.3.5 Implementation of MChannels

The implementation of MChannels is similar to Ports in Distributed Haskell [VF01].

Communication is implemented using a standard sockets library, thus avoiding the need for any extra libraries like PVM or MPI. Haskell objects are serialised using the packing routines explained before and converted into an array of bytes that can be easily communicated through a socket using TCP/IP.

The channel data type is a simple Haskell data type that contains internally all the information that will be needed for communication, i.e., the name of the channel, the name of the host where it belongs and a Concurrent Haskell channel (CHC) through which the communication between the program and the mobile runtime system occurs. When a new MChannel is created also a CHC is created to serve as a communication link between the program and the communication layer of the RTS. When a value is written into a MChannel, it is in fact written into its CHC. When the MChannel is a reference to a remote channel, the data is serialised and communicated to the appropriate host based on the information present in the MChannel data type. When the RTS receives a value from a remote host this value is written into the CHC that represents the MChannel that should receive the message. A thread that reads a value from an MChannel is in fact reading a value from the internal CHC and will stay blocked in this CHC until a value is written by the RTS there.

To make MChannels visible to other machines in the network we use the `registerMChannel` and `lookupMChannel` primitives. These primitives communicate with an external naming service that keeps listening for requests on a specific port. This service maintains a table with all the MChannels registered in the machine in which it is running. It also communicates with lookups launched by other hosts looking for channels. When a lookup is received, all the information about the channel is sent back to the client, so the client can communicate directly with the program that is waiting for requests on that channel.

5.3.6 *m*Haskell's performance

Preliminary measurements show that *m*Haskell's current implementation can be up to 20 times slower that than fast optimised implementations of mobile languages like Jocaml, and an order of magnitude slower than Java. The main reason for that is the

routine that recursively traverses the graph, forcing the evaluation of thunks before packing. Every time a computation is sent, the graph has to be traversed twice: once to force the evaluation and once for packing. It is not an option to force the evaluation while packing because the evaluation of the graph might change what has been already packed. As Jocaml is a strict language, the evaluation of expressions to be communicated occurs naturally. Moreover, Jocaml is built as an extension to the Objective Caml compiler [OCa06], a compiler with fast built-in primitives for serialisation.

*m*Haskell's implementation is still in its early stages and a lot of optimisations could be performed. For example, in the program used in the experiments, the same function is sent to different hosts and is repacked every time it is communicated. Such packed computations could be stored for reuse.

5.4 Related Work

5.4.1 The π-Calculus

MChannels are similar to π-calculus channels but with some differences. The main difference is that MChannels are single reader channels, and only threads running on the location where a channel was created, can read values from it. So, an MChannel can have many senders but only one receiver. The main reasons for this decision, as explained in section 5.1.6, are security, scalability and fault tolerance.

In the π-Calculus, if two processes share a channel, both can read from it, but only one will succeed in reading a message sent by a third process:

$$x?v \ \rightarrow \ P \mid x?v \ \rightarrow Q \mid x!a$$

As only one value is sent through x, only one expression, P or Q, will be executed. The π-calculus is a mathematical model of the changing connectivity of interactive systems, and it does not take into account the existence of locations. Basically it only models concurrency, and for that matter channels are only locations in memory that are shared by concurrent processes, although the model provides many of the intuitions needed in a distributed system, as explained in Chapter 2.

With multiple reader channels one process can *pretend* to be a server and steal messages that are not supposed to be received by him. Its a classical problem of the untyped π-calculus:

Suppose we have the following processes:

new p **in** $(P|C_1|C_2)$

where P is a printer accessible via p, and C_1 and C_2 are two clients:

$C_1 = p!j_1 \rightarrow p!j_2.()$

$C_2 = b!p \rightarrow ()$

Looking at C_1, it seems that it sends jobs to be printed in the printer P and these jobs are received and processed in the same order in which they are sent. But that is not always true because we could have a malicious process C_3:

$C_3 = b?x \rightarrow x?j \rightarrow ()$

that receives the name p from C_2 and pretends to be the printer P. To solve this problem we need to make sure that only P has reading capabilities in p, and that all the other clients are only allowed to write values in p. This problem is solved by adding types to π-calculus. As sub-typing is not available in Haskell, different types of MChannels would be needed for reading and writing, as well as different primitives that work on the different types. For simplicity, mHaskell has only one type of channels, keeping the set of new primitives as small as possible.

The main objective in designing the mHaskell primitives was to make them simple and expressive enough to be used as building blocks for higher-level abstractions. They should be similar in abstraction to the mobility calculi primitives but with a semantics realistic enough to be implemented efficiently in a real system, and scale on global networks. That is why the operational semantics for MChannels is very low level, although is still almost as simple as the π-calculus semantics.

Multiple-readers channels can be easily simulated in mHaskell. Figure 5.46 shows a simple implementation of π-calculus like channels using a home server, as described in Section 5.1.6. When a `PIChannel` is created, using `newPIChannel`, it is represented by two MChannels, one used for communication with remote references that want to

```
data PIChannel a = PIC {
                   local ::(MChannel a),
                   remote :: (MChannel (MChannel a))
                   }


newPIChannel :: IO (PIChannel a)
newPIChannel = do
                 mch <- newMChannel
                 com <- newMChannel
                 forkIO (handleRequests mch com)
                 return (PIC mch com)
      where handleRequests mch com= do
          respch<- readMChannel com
          v <- readMChannel mch
          writeMChannel respch v
          handleRequests mch com

writePIChannel :: PIChannel a -> a -> IO ()
writePIChannel (PIC mch com) v = writeMChannel mch v

readPIChannel :: PIChannel a -> IO a
readPIChannel (PIC mch com) = do
            if (isLocal mch) then
                  readMChannel mch
               else do
                  resp <- newMChannel
                  writeMChannel com resp
                  readMChannel resp
```

Figure 5.46: π-Calculus Channels implemented in *m*Haskell

read from the channel (`com`), and another in which the values are stored locally (`mch`). The `handleRequests` thread reads from `com`, requests for reading the channel. Every request is answered by reading from local channel `mch` and sending the value to the remote location. Hence, the real channel stays in the host where `newPIChannel` is called, and requests for reading the channel are forwarded to that host. The `writePIChannel` function simply calls `writeMChannel` in order to insert values into the channel: it does not matter if the channel is not local as MChannels can have multiple locations writing to it. The `readPIChannel` function writes a value into `mch` if it is called in the location where it was created, otherwise it asks the `handleRequests` thread to read the value and send it back in a temporary channel `resp`.

5.4.2 Mobility in Haskell Extensions

This section compares some of the Haskell related languages with the concepts of mobile languages presented in Chapter 2.

Eden [BLOMP97] 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 represent 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 [KOMP98]. Eden uses a closed system model with location independence. All values are evaluated to normal form before being sent through a port.

Eden and GpH (presented in Chapter 3) are simple 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) [THLP98] and Concurrent Haskell [PJGF96]. GdH allows the creation of impure side effecting I/O threads using the `forkIO` primitive of Concurrent Haskell but it also has a `rforkIO` function for remote thread creation (of type `rforkIO :: IO () -> PEId -> IO ThreadId`). The `rforkIO` function 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.

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 (as explained

in Chapter 5, these primitives can be implemented in terms of MChannels). 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. Furthermore, in the current implementation, the code for all remotely started computations must be already present in the remote locations.

Haskell with ports or *Distributed Haskell* [FH01] 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, and used using the following commands:

```
newPort      :: IO (Port a)
writePort    :: Port a -> a -> IO ()
readPort     :: Port a -> IO a
registerPort :: Port a -> PortName -> IO ()
```

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 first-order values, or types that can instantiate the `Show` class. This means that no functions or IO computations can be communicated, and the reason for this restriction is that the values of the messages are converted into strings in order to be communicated [FH01].

5.4.3 Functional Mobile Languages

Nomadic Pict Language

The Nomadic Pict Language [Woj00] extends the Pict [CT97] language with the abstractions provided by the Nomadic π-calculus [Uny01].

The language, as the calculus, provides abstractions for the creation and migration of agents:

agent a = P **in** Q agent creation
migrate to s P agent migration

As in the calculus, **agent** a = P **in** Q spawns a new agent a on the current site with body P and, after the creation of the agent, Q commences execution in parallel with the agent. The **migrate to** s P primitive makes the whole agent migrate to site s. After migrating, it starts the execution of P.

Channels, in the Nomadic Pict Language, only allow the communication of first-order values like names, tuples and basic types. Communication between agents using channels only occurs if the agents are in the same site, otherwise the agent has to migrate to the site where its partner is located. The primitive used for interaction between agents is **iflocal**, where the execution of **iflocal** <a>c!v **then** P **else** Q in the body of an agent b will send a message v to agent a, if the agents are in the same site, and P will commence execution in parallel with the rest of the body of b. Otherwise the message will be discarded, and Q will executed as part of b.

Nomadic-Pict [Woj00] was designed and implemented in Pawel Wojciechowski's Thesis, to study the problem of location independent communication between agents in a distributed system. Location independent communication is provided by a higher-level primitive:

c @ a ! v

that sends message v to channel c located at agent a. The higher-level primitive is implemented using the agent primitives, which are called *low-level* primitives.

*m*Haskell uses a completely different approach to model mobile computation than

Nomadic Pict. We start with low-level primitives that operate on Higher-Order channels, allowing the communication of any value of the language. Hence, *m*Haskell supports only *weak* mobility, while the only mobility supported by Nomadic Pict is *strong*, as two mobile agents can only communicate through channels if they are on the same location. In many cases, it can be very inefficient to migrate a whole agent to a location if it only wants to communicate the result of a computation back, specially when dealing with devices with limited resources, e.g., portable computers. Furthermore, in the next two chapters, we show that Higher-order channels such as *m*Haskell's MChannels can be used to implement higher-level primitives for mobility i.e., remote evaluation, strong mobility and location independent communication.

Jocaml

Jocaml [CF99] is an extension to Objective-Caml [OCa06] used to program systems with mobile agents. Jocaml extends Objective-Caml with a small set of primitives taken from the Join-Calculus [FG96]. The model is based on *locations* that gather both agents and sites in a single abstraction, and *channels*, which are communication links kept during migration of locations.

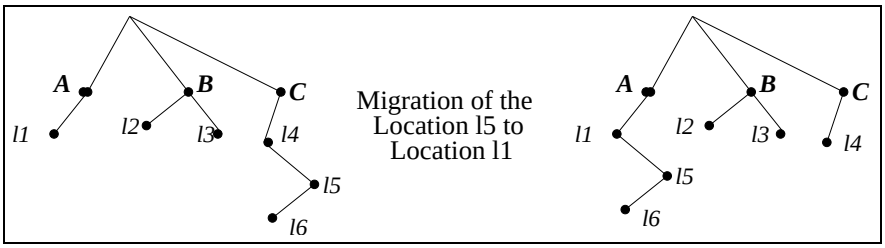


Figure 5.47: Location Tree and Migration

The organisation of the locations can be seen as a tree (Figure 5.47): sites are top-level locations which are created at the root of the tree (locations A, B and C in figure 5.47), whereas agents are nested locations, i.e., locations which are created inside other locations. As a consequence the migration of a location moves together all its children in the tree.

A construct called *definition* is used to create channels and processes. When a message is received on a channel, a new thread is created to execute the associated process. All communication links are kept during migration and channels can be transmitted between agents in messages. Channels can be *synchronous* or *asynchronous*.

Channels are links between one receiver and multiple senders. A simple *synchronous* channel is defined as:

$$\text{let def } name(\text{mesg}) = P(\text{mesg})$$

It creates a channel called *name* and a receiver that will create a new thread in the agent to execute P for each message *mesg* received. Asynchronous channels are defined by suffixing the channel name with !. Using *join-patterns*, is also possible to express synchronisation in a channel definition. In the following example:

$$\text{let def } v!(\text{a}) \mid g!(\text{b}) = P(\text{a,b})$$

the process P will only be executed if there are messages in channel v and in channel g. In Jocaml, sites, agents and groups of agents are represented by an abstraction called *location*:

$$\text{let loc } agent \text{ do } process$$

This example creates a new agent named *agent* executing the thread *process*.

Remote Location:

```
loc remote end
do ns.register (("remote"), ushas.cee.hw.ac.uk)
```

Migrating Client:

```
loc client
  init
    let remote = ns.lookup("remote",ushas.cee.hw.ac.uk);
    in go (remote); print_string("Hello Remote Host");
  end
```

Figure 5.48: Simple mobile program in JoCaml

In figure 5.48, a simple mobile program written in JoCaml is presented. First a remote location is created and registered in a name server, that is running on a machine called `ushas`, using the `ns.register` command. Then a simple client program that looks for a reference to the remote location in the name server and migrates to it, where a string is printed.

Having locations that move, creates the same problem as having multiple reader channels. Although channels in Jocaml are single reader, if a location where a channel

was created moves to a new location, all references to this channel must be able to find it transparently. The channels provide location independent communication, and a complicated infrastructure is necessary to support them. The migration of nested locations also needs complicated algorithms to be implemented, to avoid race conditions between the locations. The infrastructure necessary for such a high-level infrastructure is difficult to be implemented in large scale networks. Furthermore, in Jocaml, there is only one name server running for the whole distributed system. This server accepts the registration of locations, channels and resources. If the name server is running on host A, and a program running on B, migrates to C to use a local resource on C, it still has to query the server in A to access the resource. Such centralised server can be a bottleneck in larger applications for global networks, and a problem with the name server can compromise the whole system.

Facile

Facile [GMP89] is a distributed functional language that extends Standard ML with primitives for concurrency, distribution and imperative programming. In [Kna95], Knabe extends the Facile language and compiler to support mobile computation. The mobile version of Facile can be seen as a weakly mobile language that supports the communication of higher order values through channels, similar to *m*Haskell, although in *m*Haskell the communication primitives are kept in the IO Monad. In Facile the programmer has to annotate all the potentially transmissible functions, using the `xfun` keyword, so that the compiler knows that it has to generate transmissible representations for them. Making the programmer to annotate transmissible functions has some disadvantages. Sometimes it is difficult to find exactly which functions are going to be communicated, especially when higher order functions are used in the expressions. If a function that is going to be sent through a channel is not annotated, a runtime exception will be raised. With this in mind, programmers may choose to annotate more functions than is necessary. In *m*Haskell, functions are already compiled into an architecture independent representation, the BCO, hence no annotations are needed, and modules in machine code are considered ubiquitous and are not communicated. Another difference between Facile and *m*Haskell is in the way resources are accessed. In Facile, the programmer has to implement proxies that contain the types of remote resources that will be accessed by the mobile computation, and the mobile program,

during compilation, is typed checked against these proxies locally. If the mobile program then migrates to the remote location and the type of the remote resource happens to be different from the proxy, or the remote resource doesn't exist, a runtime exception will be raised. In *m*Haskell, the types of resources can be typed checked at runtime, when the mobile computation arrives in the remote location, using dynamic types. If the types don't match, the mobile program can recover by migrating to a new location or returning home.

Other Functional Mobile Languages

There are other extensions to functional languages that allow the communication of higher-order values. Kali-Scheme [CJK95] and Erlang [Erl06] are examples of strict weakly typed languages that allow the communication of functions. Haskell is a statically typed language hence the communication between locations can be described as a data type and many mistakes can be caught during the compilation of programs.

Curry [Han99] is a functional logic language that provides communication based on Ports in a similar way to the extension presented in this Thesis. Goffin [CGK98] is a Haskell extension for concurrent constraint programming using ports but there is no distributed implementation of the language available yet. Another language that is closely related to our system is Famke [vWP02]. Famke is an implementation of threads for the lazy functional language Clean [NSvEP91] (using monads and continuations), together with an extension for distributed communication using ports. Famke has only a restricted form of concurrency, providing interleaved execution of atomic actions using a continuations monad.

Comparison

We argue in the Introduction of this Chapter that a language for mobile computation should provide a small set of primitives for mobility, that are similar to process calculi primitives, with a simple semantics, and that these primitives should be expressive enough in order to model the main mobility dialects, including high-level abstractions for mobility. MChannels are a Haskell extension for communication that is different from any other Haskell extension as they allow the communication of any Haskell value on open networks, including functions and `IO` computations. MChannels are similiar to π-calculus channels (see Section 5.4.1), and have a simple operational semantics that

Table 5.6: Comparing Functional Languages for Mobile Computation

	Jocaml	Nomadic-Pict	Kali	Facile	mHaskell
Weak Mobility	yes	no	yes	yes	yes
Strong Mobility of Threads	Migration of Locations	yes	yes	no	yes (Chap 6)
Open Networks	no	yes	no	yes	yes
High-Level Abstractions	no	Location independent Communication	no	no	yes mobility Skeletons (Chap 7)
Formal Foundations	Join Calculus	Nomadic π-Calculus	??	yes	π-Calculus
Statically Typed	yes	yes	no	yes	yes
Purely Functional	no	no	no	no	yes

reflects how its implementation should work (see Section 5.2). MChannel are a small, low level extension for mobility and programming with them can be seen as system level programming. The objective of MChannels is to provide a collection of primitives that can be used to implement, as described in Chapter 6, common abstractions for mobility such as *weak mobility* using remote evaluation, or strong mobility using a *moveTo* construct, and new high level ones such as *mobility skeletons* that are higher order functions that encapsulate common patters of mobile computation.

While some mobile languagens like Nomadic-Pict only provide strong mobility, *m*Haskell in its core, only provides weak mobility, but strong mobility can be implemented, as another layer in the system, using the low level primitives as described in Chapter 6. MChannels can also be used to implement common weak mobility primitives for the creation of remote threads, as the ones available in Kali Scheme and GdH(see `reval` and `rfork` in Section 6.1). As stated in Chapter 2, an explicit mobile language should be designed to work on large scale systems and *m*Haskell was designed to be a mobile language that operates on open networks where locations can join and leave the computation at any time. Hence, its runtime support for mobility includes name servers at each *m*Haskell node, avoiding centralized servers that would introduce a single point of failure like in Jocaml. Other features like mobility of locations, and location independent communication, present in some mobile languages, need expensive algorithms to be implemented that would not scale in large systems and therefore are not provided in the core *m*Haskell implementation. But, as described in Chapters 6 and 7, such features can be modeled in *m*Haskell using MChannels or weak mobility.

*m*Haskell is a purely functional language where stateful computations are kept in the `IO` monad, therefore there is a clean separation between computations with side-effects and stateless functional code. Furthermore the purely functional part of the language can be used to implement new abstractions for statefull actions as `IO` computations are first-class values in Haskell. Haskell's strong type system ensures that `IO` actions will only be executed in the right context which helps reasoning about computations with effect, including mobile computations.

5.5 Summary

Mobile Haskell (*m*Haskell) is an extension of the purely functional Haskell language designed for the implementation of distributed mobile software. *m*Haskell extends Concurrent Haskell [PJGF96], an extension supporting concurrent programming, with higher order communication channels called *Mobile Channels* (MChannels), that allow the communication of arbitrary Haskell values including functions, IO actions and channels.

*m*Haskell *supports the construction of open systems*, enabling programs to connect and communicate with other programs and to discover new resources in the network. Computations communicated using *m*Haskell provide stand alone execution: there are no implicit call-backs to the locations that launched the mobile application. Mobile programs are compiled into byte-code and modules that are in machine code, e.g., libraries or the primitives of the language, are considered ubiquitous and are not communicated.

MChannels can be seen as low level mobile constructs for system level programming. In the next two Chapters we show how *m*Haskell can be used to implement higher-level abstractions for mobility and mobile applications.

Chapter 6

Coordination Abstractions for Mobile Computation

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 6.7. At the lowest level the programmer explicitly controls all aspects of coordination using primitives such as `send` and `receive` for communication. Mid-level abstractions encapsulates several coordination aspects into a single construct, e.g., a Remote Method Invocation (RMI) [Gro01] encapsulates communication from the client to the server, execution of a server method, and communication back to the client. High level abstractions aim to control many 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 [Lov93], or algorithmic skeletons [Col89]. 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 [Arm03], and design-patterns for distributed computing [GHJV95] available for object oriented languages.

Programming with MChannels can be seen as low-level system programming, as the

Table 6.7: 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` and `moveTo`
Low Level	`send`, `receive`	`send`, `receive`	`send`, `receive` (MChannels)

programmer has to worry about all aspects of coordination. Some of the main advantages of functional languages are the ease of composing computations and the powerful abstraction mechanisms [Hug89]. Stateful computations in Haskell are first class values in the language hence they can be manipulated and new high level abstractions over computations can be defined.

In this Chapter, we show how the low level MChannel primitives can be used to implement high level abstractions for mobility. First, Section 6.1 shows how MChannels can be used to express weak mobility constructs for the creation of remote computations i.e., remote evaluation and remote thread creation. We have used these mechanisms to develop parameterisable higher-order functions in *m*Haskell that encapsulate common patterns of mobile computation, called *mobility skeletons*, one of the first high-level abstractions proposed for mobility. These skeletons hide the coordination structure of the code, analogous to algorithmic skeletons [Col89] for parallelism. Mobility skeletons abstract over mobile stateful computations on open distributed networks, and they can be nested and composed to describe different behaviours.

Mobile languages that allow the migration of running threads are said to support strong mobility. In mobile languages that have native support for continuations (through a construct like `call/cc` [FF95]), a strong mobility construct is easily implemented by capturing the continuation of the current computation and sending it to be executed remotely. The current implementations of Haskell do not have built-in support for continuations, but it is well known how continuations can be elegantly implemented in Haskell using a *continuation monad* [Wad95, Cla99]. 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. Section 6.3 shows how strong mobility can be implemented in a language like *m*Haskell using higher-order channels and a continuation monad.

6.1 Medium Level Coordination

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`).

6.1.1 Remote Thread Creation

In *m*Haskell, a thread can be created in a remote location with the `rfork` function, analogous to `forkIO` in Concurrent Haskell:

```
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. The operational semantics of `rfork` is a single extension of the semantics in Section 5.2, as shown in Figure 6.49.

$$L(s, \{\mathbb{E}[\texttt{rfork}\ M\ s']\}_t) \rightarrow L(s, \{\mathbb{E}[\texttt{return}\ ()]\}_t)\ (s', \{\mathbb{E}[V']\}_{t'}) \quad (RFORK)$$
$$\text{where } V' = \texttt{forceThunks}\ M$$

Figure 6.49: Transition rule for `rfork`

The `rfork` function can be implemented using MChannels, as in Figure 6.51. The *m*Haskell `rfork` implementation relies on every location executing a remote fork server, depicted in Figure 6.50. The `startRFork` server creates a channel with the name of the location where it executes and then repeatedly reads computations from the channel and forks local threads to execute them.

The `rfork` function looks for the channel registered in the `startRFork` server, that is a channel with the same name as the remote location (`lookupMChannel host host`), and sends the computation to be evaluated on the remote location `host`.

Note that `rfork` is an asynchronous operation: it sends a computation to be executed remotely but does not wait for its execution, as can be seen in the semantics

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

Figure 6.50: The remote fork server

```
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"
```

Figure 6.51: The `rfork` function

(any work in the set modelling threads can be executed next); this behaviour is directly reflected in the implementation.

6.1.2 Remote Evaluation

In the *remote evaluation* [Vol96] paradigm, a computation is sent from a host A to a remote host B to use B's resources. It is straightforward to implement *remote evaluation* using mobile channels and `rfork`. A computation can be sent to be evaluated on a remote location using the `reval` function:

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

Remote evaluation is a synchronous operation, it sends a computation to be executed on a remote location and only returns when the computation has finished. The `reval` function, Figure 6.52, uses `rfork` to execute the expression (`job >>= \r -> writeMChannel mch r`) on the remote location. The expression executes `job` and sends the result of its execution back to the first location through the MChannel `mch`.

```
reval :: IO a -> HostName -> IO a
reval job host = do
        mch <- newMChannel
        rfork (job >>= \r -> writeMChannel mch r) host
        result <- readMChannel mch
        return result
```

Figure 6.52: The implementation of `reval`

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 be blocked waiting for the result of `reval`.

The program that computes the total load of a network, given in Figure 5.37, can be shortened using `reval`, as in Figure 6.53.

```
main = do
  list <- mapM (reval mobile) listofmachines
  let v = sum list
  print ("Total Load of the network: " ++ (show v))
  where
   mobile =  (...)
```

Figure 6.53: Shortened version of the program that computes the load of a network

The abstraction provided by `reval` is similar to that provided by Java™'s RMI, the difference being that `reval` sends the whole computation (including its code) to be evaluated on the remote host, and RMI uses proxies (stubs) in order to give access to remote methods.

6.2 High Level Coordination: Mobility Skeletons

This section identifies three common patterns of mobile computation and implements them as higher order functions, or *Mobility Skeletons* in *m*Haskell, using `reval` and `rfork` from Section 6.1.

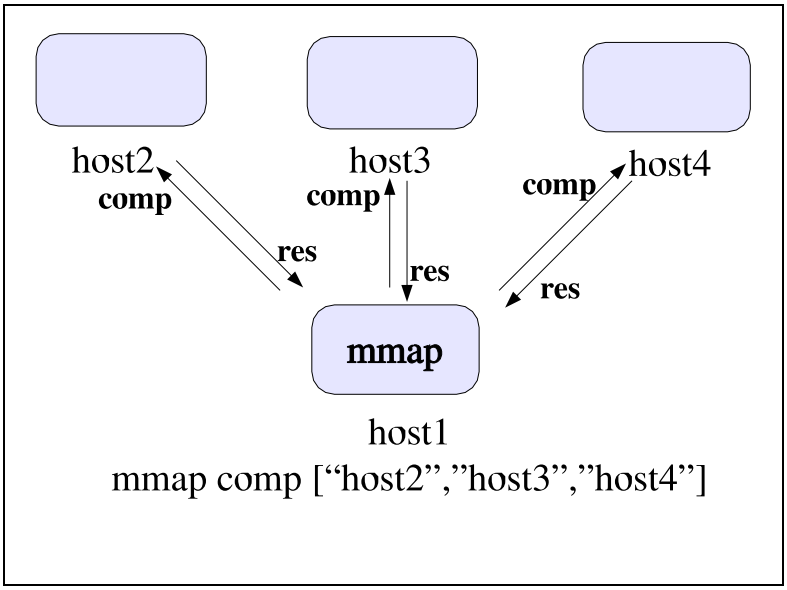


Figure 6.54: The behaviour of `mmap`

Mobility skeletons are analogous to algorithmic skeletons [Col89] 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 typically anonymous locations. In contrast mobility skeletons abstract over stateful (or impure) computations in an open network, i.e., a dynamic set of named 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` uses the stateful nature of the computation to repeatedly probe the state at different times and make decisions based on the result.

In this section we describe synchronous and asynchronous versions of three mobility skeletons: `mmap`, `mfold` and `mzipper` and give simple examples using them. In Chapter 6, we use the skeletons to implement a larger application: a distributed meeting planner.

6.2.1 `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 6.54) 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.

```
mmap::IO b -> [HostName] -> IO [b]
mmap f hs = mapM (reval f) hs
```

Figure 6.55: The definition of the `mmap` skeleton

```
mmap_::IO() -> [HostName] -> IO()
mmap_ f hs = mapM_ (rfork f) hs
```

Figure 6.56: The definition of the `mmap_` skeleton

The implementation of `mmap` (see Figure 6.55) executes a remote evaluation of its argument `f` on every host of the list `hs`. It uses `mapM` to apply and execute `reval f` on all the locations of `hs`. For some examples, it is useful to have a simpler asynchronous version of `mmap`, that broadcasts the action to the locations, without waiting for any results, this version is called `mmap_` (see Figure 6.56).

The `mmap` skeleton encapsulates the pattern of coordination present in two applications described earlier, see Figures 5.37 and 6.53, that determine the load of all locations in a network. Here we give the same program using `mmap`:

```
networkLoad :: [HostNames] -> IO [Int]
networkLoad hosts =  mmap getLocalLoad hosts
```

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 in Figure 6.57. This function, when called on a location, looks for a resource named `"getLoad"` and executes it, returning the load of the current location.

6.2.2 `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 6.58). This pattern matches the concept of a distributed information retrieval (DIR) system. A DIR application gathers information matching some specified criteria from information sources

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

Figure 6.57: The `getLocalLoad` function

dispersed in the network. This kind of application has been considered "the killer application" for mobile languages [FPV98].

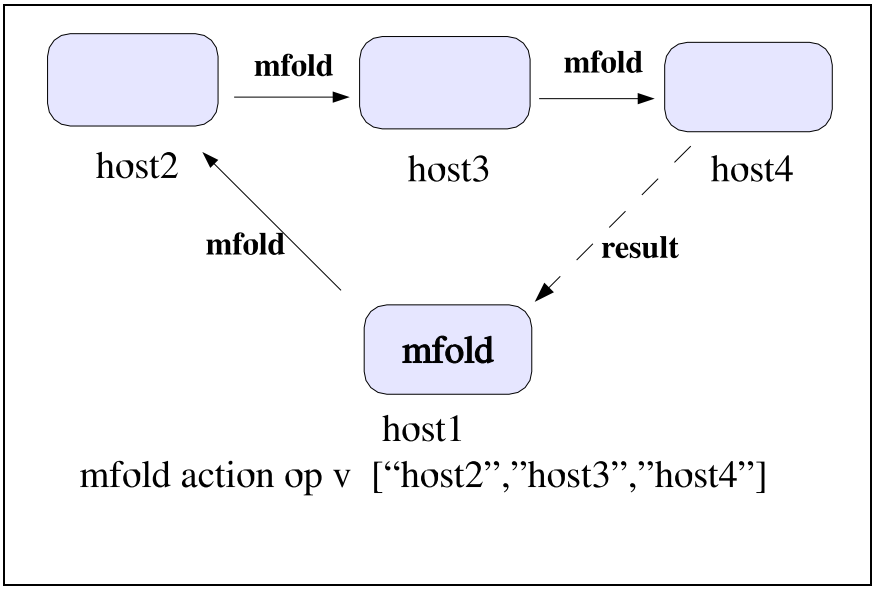


Figure 6.58: The behaviour of `mfold`

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

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

It 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 list of locations to visit, and returns the result of combining the values of type `a`. 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`.

The synchronous `mfold` skeleton (Figure 6.60) is implemented using a more general asynchronous skeleton called `mfold_` (Figure 6.59). As can be seen in Figure 6.59, `mfold_` takes an extra argument that specifies what should be done with the result of the computation once the program has visited all the hosts in the list: if the list of

```
mfold_ :: IO a -> (a -> a -> a) -> a -> (a -> IO ()) -> [HostName] -> IO ()
mfold_ f op v final [] = final v
mfold_ f op v final (h:hs) = rfork (code f op v final hs) h
 where code f op v final hosts = do
        v2 <- f
        mfold_ f op (op v v2) final hosts
```

Figure 6.59: The definition of the `mfold_` skeleton

```
mfold :: IO a -> (a -> a -> a) -> a -> [HostName] -> IO a
mfold action op v hosts = do
   mch <- newMChannel
   mfold_ action op v (\x -> writeMChannel mch x) hosts
   readMChannel mch
```

Figure 6.60: The definition of the `mfold` skeleton

locations to be visited is empty, then it simply applies the extra function to the result. If the list of locations to visit is not empty the computation `code` is executed on the head of the list. The function `code` executes the action `f` on the current host and then does a recursive call to `mfold_`, combining the current value with the result from the execution of `f`. In the implementation of `mfold`, the extra function in `mfold_` sends the result of the computation back to the host that called `mfold` through an MChannel

The `mfold` skeleton can also be used to construct an application that computes the total load of a network. Using the `getLocalLoad` function defined in the previous section, `totalLoad` can be implemented as follows:

```
totalLoad :: [HostName] -> IO Int
totalLoad hosts = mfold getLocalLoad (+) 0 hosts
```

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

The behaviour of the program can be easily modified by changing the arguments passed to `mfold`. For example, this program:

```
list <- mfold (getLocalLoad >>= \x-> return [x]) (++) [] 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`, the skeletons have 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 6.54 and 6.58).

The `mfold` skeleton is very different from 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.

6.2.3 `mzipper`: Iteration

Another pattern of mobile computation is a computation that visits a sequence of locations, looking for some value common to all locations. The value is tested against a predicate on every location, and if it fails, the computation returns to the start of the sequence of locations trying a new value, as depicted in Figure 6.61.

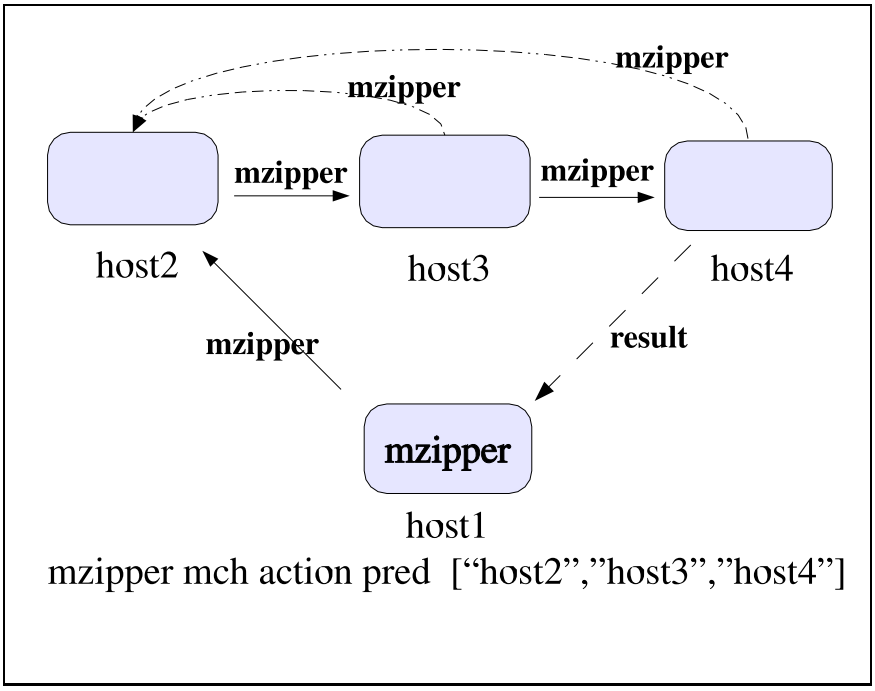


Figure 6.61: The behaviour of `mzipper`

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` [Hue97], that describes how to navigate on different data structures.

The `mzipper` template has the following type:

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

It takes as arguments a function that receives a list of values in which the locations disagreed in the past and returns a new value, a predicate that indicates if the current location agrees with the current value, a list of locations to visit, and returns the final agreed value, if there is one. The `mzipper` skeleton is also implemented using an asynchronous skeleton called `mzipper_`, that takes an extra argument that tells what it should do with the result once the last host is reached.

The implementation of the `mzipper_` skeleton can be seen in Figure 6.62. The objective of the first part of the definition is to create a first value to be tested on all locations. It starts by checking if the first location to be visited is the current location. If the current location is not the first element of the list then `mzipper_` is started again, with the same arguments, on the head of the list. Otherwise, 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 agreed. If a value is not found, it returns `Nothing` to the action `final`, that should tell the skeleton what to do with the result. Otherwise the recursive zipper function is called (`rmzipper`), which takes two extra arguments, the list and the value.

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.

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

Figure 6.62: The definition of the `mzipper_` skeleton

The synchronous `mzipper` skeleton (Figure 6.63) is implemented in a similar way to `mfold`.

As an 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
     res <-  mzipper (myThreshold threshold) isBelowTh hosts
```

```
mzipper :: ([a] -> IO ([a],Maybe a)) ->
                    (a -> IO Bool) -> [HostName] -> IO (Maybe a)
mzipper action pred hosts = do
     mch <- newMChannel
     mzipper_ action pred (\x -> writeMChannel mch x) hosts
     readMChannel mch
```

Figure 6.63: The definition of the `mzipper` skeleton

```
     case res of
       Nothing -> isLoadBelow threshold hosts
       Just x  -> return True
 where
 myThreshold = (...)
 isBelowTh = (...)
```

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. It uses uses `mzipper` to check if all the locations in the network have the appropriate load. Every time `mzipper` returns `Nothing` through the MChannel `mch`, `isLoadBelow` 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 `isLoadBelow` function has to call itself again every time `mzipper` returns `Nothing`.

6.2.4 Nesting and Composing Skeletons

One of the advantages of using a functional language to implement the mobility skeletons, is that it facilitates composing and nesting 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 `totalLoad`, 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<- totalLoad l
             return r
```

Then, to compute the load 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 `totalLoad` function with `mmap_`:

```
getLoadAndBC hosts = totalLoad hosts >>= \ load -> mmap_ (update load) hosts
```

or using the `do` notation:

```
getLoadAndBC hosts = do
    load <- mgetLoad hosts
    mmap_ (update load) hosts
```

where `update` updates a resource in all the `hosts` 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.

6.3 Strong Mobility

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 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 Jocaml [CF99] system). These routines for serialisation are more complicated than those usually available in programming languages (e.g., Java), as not only data must be communicated but the state of the whole computation including registers, stacks and memory. The work on thread migration presented in Chapter 3, 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` [FF95]) 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 [CJK95]). In some languages that do not have native support for continuations, strong mobility is implemented 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* [Sek99] and Klaim [BN01].

In this Section, we present a somewhat different means for implementing 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* [Wad95, Cla99], 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 6.3.1 a new type of *mobile threads* based on a continuation monad is presented. Section 6.3.2 describes how a primitive for strong mobility can be implemented using weak mobility and the continuation monad. In Sections 6.3.3 to 6.3.5 examples of the use of strong mobility are given, including a tree search algorithm and a new implementation of `mfold_` using `foldr` and lazy evaluation.

6.3.1 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 [FF95]. To make the continuation of the current thread available in Haskell, we use a simple continuation monad, adapted from [Cla99]:

```
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
```

where the `$` operator is an infix version of function application, used to eliminate the use of parentheses (`f $ x = f 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 of the continuation monad. In Haskell, threads created using concurrent Haskell are executed inside of the IO monad, hence mobile threads should be able to execute IO actions in a similar way. The `atom` function, takes any IO action (of type `IO a`) and transforms it 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 (a good tutorial on monad transformers is [New06]):

```
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 computation. In the case of the continuation monad, it will execute the `Action`s, 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 RTS. 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.

6.3.2 The `moveTo` operation

The `moveTo` function, that appears in Figure 6.64, sends the continuation of the current thread to be executed on a remote host, and terminates the thread that called it.

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

Figure 6.64: The `moveTo` function

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 finishs after sending its continuation for remote execution.

6.3.3 Example 1: Simple strong mobility

In Figure 6.65 a simple example using strong mobility is given. I starts a mobile thread, using `forkMT`, which gets the name of the current location, and moves to a new location where the name of the previous location is printed.

```
main = forkMT ex


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

Figure 6.65: Simple Strong Mobility Example

As `getHostName` (of type `IO String`) and `print` (of type `String -> IO ()`) are actions in the `IO` monad, they have to be lifted into the continuation monad using `lift`. 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.

6.3.4 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 example, taken from [Sek99], consider the *m*Haskell program in Figure 6.66. 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 execute 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.

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

Figure 6.66: Tree search using strong mobility

In the base case, when `mTreeSearch` 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. For each branch, it does a recursive call to `mTreeSearch` to search for a `Leaf`.

One could try to write the same recursive program using remote evaluation as in the example of Figure 6.67. Looking closely at 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 6.68, where there is a node `A` with two subtrees `B` and `C`. The program in Figure 6.66 would migrate from `A` to `B`, and then to `C`. The program using remote evaluation (Figure 6.67) migrates from `A` $\rightarrow$ `B` $\rightarrow$ `A` $\rightarrow$ `C` $\rightarrow$ `A`. While the addition in the first program takes place in `C`, in the second it takes place in `A`.

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 [Sek99]. Hence, the advantage of using mobile threads and the `moveTo` construct, is that the

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

Figure 6.67: Tree search using weak mobility

continuation is hidden in the continuation monad, and the program can be written as a normal Concurrent Haskell program.

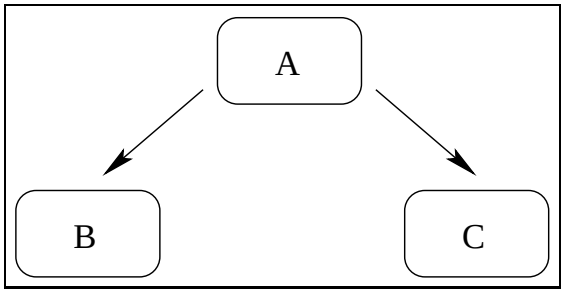


Figure 6.68: Searching in a Tree

6.3.5 Example 3: mfold

As a final example, a different implementation of `mfold_` is presented in Figure 6.69.

It has the same behaviour as the implementation presented in Section 6.2.2, but uses a sequential `foldr`. It is interesting to notice that although the function `move` is mapped over the list of `hosts`, because of lazy evaluation, the mobility of the thread only occurs when the `foldr` consumes the list.

The application of `mfold_` to its arguments, generates a mobile thread, hence the program must be executed using `forkMT`, as in the implementation of `mfold` of Figure 6.70. The `mfold` function is blocked reading from the MChannel `mch` until the mobile thread sends the result of its computation.

6.4 Summary

In this Chapter we have demonstrated that the low level MChannel primitives are expressive enough to implement both common mobility constructs for weak mobility (i.e., `rfork`, and `reval`), and strong mobility (`moveTo`), as well as new high level abstractions

```
mfold_ :: IO a -> (a -> a -> a) -> a -> (a -> IO ()) ->
            [HostName] -> M IO ()
mfold_ f op v final hosts = do
    result <- foldr (opm 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


opm :: (a -> a -> a) ->M IO a -> M IO a -> M IO a
opm op x y = do
        r1 <- x
        r2 <- y
        return (r1 `op` r2)
```

Figure 6.69: `mfold_` using strong mobility

```
mfold action op v hosts = do
   mch <- newMChannel
   forkMT (mfold_ action op v (\x -> writeMChannel mch x) hosts)
   readMChannel mch
```

Figure 6.70: `mfold` using a mobile thread

called *mobility skeletons*, that are higher order functions that abstract over common patterns of mobile computation. Mobility skeletons combine stateful computations using monads, allowing the programmer to use familiar notation for the code executed on each machine and retaining the semantics of the underlying purely-functional language. The skeletons presented in this Chapter, encapsulate patterns of mobile computation but *m*Haskell can also be used to write other types of skeletons similar to algorithmic skeletons, as in the thread farm presented in the next Chapter. The implementations of the skeletons presented in this Chapter are very simple, as the main objective was to catalogue and describe the patterns identified. More robust skeletons could be implemented e.g., by adding extra-arguments to the skeletons to describe what happens when things go wrong, e.g., not being able to reach one of the locations in the list of location to visit. Also, the three mobility skeletons identified encapsulate patterns of

mobile computation that usually happen in distributed information retrieval systems. It would be interesting to analyse commercial and research mobile applications trying to identify more mobile coordination abstractions that are general i.e., applicable in many cases, realistic i.e., useful for real applications, and easy to reason about.

The idea of mobility skeletons could also be applied in other programming languages. For example, in Appendix A, there is an implementation of the `mmap` skeleton as a *template design pattern* in Java.

Powerful abstraction mechanisms, such as higher order functions and polymorphic type systems, make it easier for programmers to abstract over common patterns of computation, and to write abstractions such as remote evaluation, and strong mobility. Although many mobile languages are based on the functional paradigm, as far as we know, no one tried to specify common communication behaviours in mobile programming as higher order functions.

The support for monadic programming and interaction between monads available in Haskell was very important in the development of *m*Haskell's abstractions. To support strong mobility, we implemented a simple continuation monad that can operate together with the IO monad, hence inheriting support for concurrent and distributed programming using Concurrent Haskell and MChannels. We demonstrated that strong mobility can be elegantly implemented in a language with weak mobility, higher-order channels and first-class continuations. Furthermore, IO actions are first class values in Haskell, making it easier to implement abstractions such as the creation of remote computations and skeletons.

Chapter 6 shows how the mobility constructs designed in this Chapter can be used in the implementation of real applications that use mobile computation.

Chapter 7

Mobile Applications

To evaluate the *m*Haskell language design and the abstractions presented in the previous Chapter, this Chapter shows the development of three non-trivial mobile applications: a distributed meeting planner, a distributed stateless web server and a platform for mobile agents.

In the distributed meeting planner, users launch mobile programs that visit the locations of people involved in a meeting, trying to find an empty slot in their time tables for the meeting. Section 7.1 shows that all the communication and coordination needed in this sort of application can be expressed using mobility skeletons The programmer only has to implement the application specific parts of the program, i.e., the arguments for the skeletons, that are simple sequential Haskell programs.

A Stateless server stores the continuation of its computation in the document that it sends to clients, e.g a Stateless Web Server stores all the persistent knowledge about the interaction with the client in the HTML document that is sent back to the browser. The advantage is that no connection needs to be kept between the client and the server, and the computation can be restarted at any point later, without the server having to keep any record of the previous interaction. In this Chapter, a stateless web server is implemented using the serialisation primitives described in Section 4.3.2. This shows that the technology developed for *m*Haskell can also be used for other means, .i.e., persistent storage of applications. Furthermore, the web server is made distributed by using a new skeleton, a *thread farm*, to offload tasks from a heavy loaded server to a cluster of computers.

Finally we describe a simple mobile agent platform that supports partially connected

devices, i.e., devices that are not always connected to the network, such as laptops and PDAs. Such devices can benefit from the use of mobile agents, as they can launch agents to perform tasks in a network and then disconnect, only connecting to the network later to collect the results produced by the agents.

7.1 Case Study: The Distributed Meeting Planner

The distributed meeting planner is depicted in figure 7.71. We assume that each user is connected to one workstation and, when one of them 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.

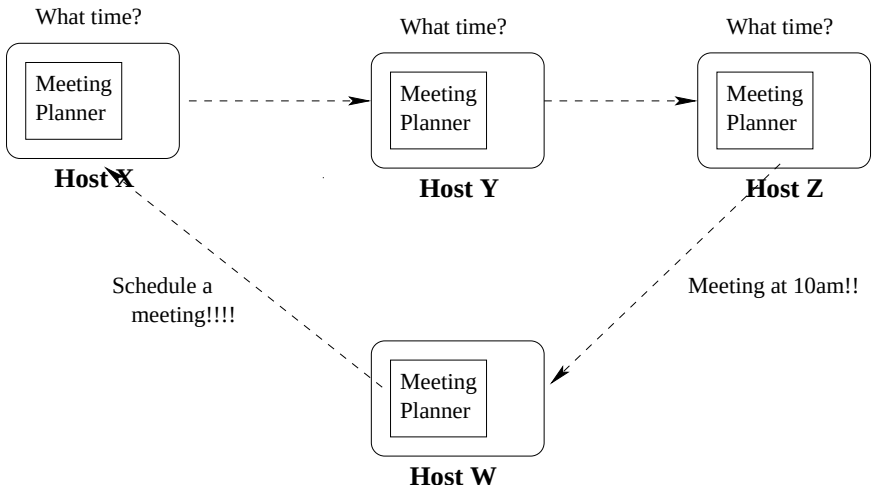


Figure 7.71: The Distributed Meeting Planner

This sort of application has two patterns of mobility. The first one is the pattern of a computation that visits a set of locations in a network performing some actions at each location, as can be seen in figure 7.71. In Chapter 5, two skeletons that perform this sort of pattern were given, `mfold` and `mzipper`. The second pattern is the idea of broadcasting a computation to a list of locations, or `mmap`. In the case of the meeting planner this pattern appears when the time of the meeting must be updated in the time tables of all the users of the system.

The next section presents a version of the program using `mzipper`, and in Section 7.1.2, some of the problems of the meeting planner are discussed and a new version using `mfold` is described.

7.1.1 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 a time (time here is represented as a `String`, although any other data type could be used), that everyone is available, if there is one.

```
timeMeeting :: [HostName] -> IO (Maybe String)
timeMeeting hosts = mzipper getNewTime timeOK hosts
```

The `timeMeeting` function uses `mzipper` to visit the locations and check for an available time for the meeting. The `mzipper` function chooses an available time from the timetable of the initiating location and then visits the other locations in the list to check if that time is available. If the time is not available at one of the locations, then `mzipper` returns to the initiating location and asks for a different time. `mzipper` will return once all the locations agreed with a time, or when the first location does not have any other time to suggest. `mzipper` is called with two arguments: the `getNewTime` function that returns an available time on the initiating location, and `timeOK` that is executed at every location to check if the proposed time is available at that location. The `getNewTime` function takes 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 has to exist 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 from last Chapter. 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`, i.e., 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. When the meeting planner is started on every location, the first action that it takes is to register its local resources. As more than one resource with different types are registered, i.e., `getFreeTimesIO` is used by `timeOK`, we need to register them as dynamic values using the `toDyn` function:

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

The second argument given to `mzipper` is `timeOK :: String -> IO Bool`, which is executed on every location to check if the current time is suitable. It looks up for a resource called `"freetimes"`, and executes it to get the free times of the current location. Then, it checks if the current time for the meeting is included in that list.

```
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
```

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, if the application has an agreed time for the meeting, it broadcasts the time to all the locations using `mmap_`, and the `updateTime` function that records the new meeting time in the timetable of a location.

```
mmap_ (updateTime time) listofhosts
```

7.1.2 Using `mfold`

In the previous implementation of the meeting planner, whenever one of the locations is not available at the proposed meeting time the computation returns to the first location and restarts the search. As every location already has a function that returns all the free times available (`getFreeTimesIO`), the program could be optimised to carry not just one free time, but a list with all the free times available in the first location. A function like `combineStrings :: [String] -> [String] -> [String]` that, computes the intersection of 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 hosts = do
          myfreetimes <- getFreeTimesIO
          mfold getLocalTimes combineStrings myfreetimes hosts
```

Now, `createMeeting` is described in terms of a `mfold`. It will visit all the locations in `hosts`, 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 -> getFreeTimesIO
```

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.

7.2 Case Study: A Stateless Web Server

7.2.1 Stateless Servers

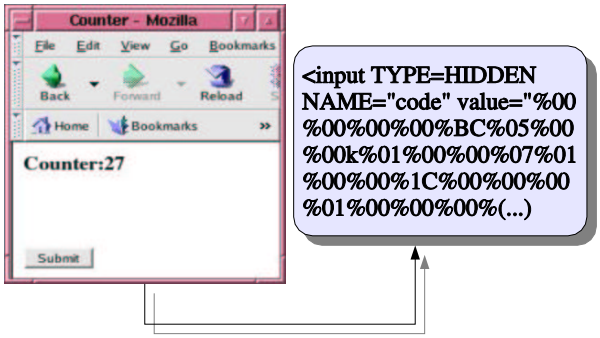


Figure 7.72: The counter example

In stateless servers [Hal97], all persistent knowledge about applications is kept in the documents exchanged between clients and servers. Servers and clients are stateless: the server stores the *continuation* of its application in the document that it sends to the client. When it receives an answer back from the client, it doesn't need to remember anything from the previous interaction, it simply executes the code with the new input. In this sense, the notion of a state is replaced by a persistent representation of requests that contain all data needed to perform the computation. Stateless servers have several advantages:

- Neither a server-side state nor a permanent connection with the server has to be maintained.
- Different servers can be used to process the code. In particular, execution of the code is not restricted to the server that stores the program the client wants to use. For example if the client is paying for the use of the server, the computation can be continued in a cheaper server.

- A computation can be continued much later. For example in a web context, the client can start interacting with a server, then bookmark one of the pages locally, and restart the computation some day later without the server needing to keep any information about the interaction. All the information, including the computation, is kept in the client.

The idea of a stateless server suits well a distributed system in which there are many lightweight clients and some powerful servers. A client could be just a simple PDA, a web client running on a cheap hardware or a workstation accessing a powerful computational GRID [FKT01]. The server needs to provide only its processing power: it receives code sent by clients, executes it and sends the result back.

Mobile languages are a perfect implementation platform for stateless servers as applications can be saved and restarted at any time. In this section, we use the serialisation primitives, described in Section 5.3.2 to implement, following the ideas presented in [Hal97], a stateless web server that keeps the state of its computations in the pages that it sends to browsers, and executes code sent in the clients requests. Furthermore, by using MChannels and a thread farm, we make the web server distributed, using a cluster of machines to process the requests sent by clients. Hence, in this example, we use *m*Haskell to implement two aspects of mobility: persistence and distribution. Web browsers do not need to be modified in order to use our web server: the code is kept in the web page as a hidden tag, no special work has to be done in the client, except sending the code back to the server in a POST message.

7.2.2 A Stateless Web Server

Following the Concurrent Haskell web server [Mar00], the *m*Haskell web server is implemented using a simple main loop:

```
connectC socket = do
        (s,add) <- Network.Socket.accept socket
        forkIO (processMsg s)
        connectC socket
```

The server repeatedly reads requests from a socket and forks threads to process these requests. The main difference occurs when one of the messages contains the code for an application, as in Figure 7.73. In this case, we use the `getCode` function, that

```
case (check msg) of
(...)
 Code -> do
     (codeBuffer,args) <- getCode msg
     computation <- unpackV codeBuffer
     response <- computation args
     sendResp socket response
```

Figure 7.73: Receiving code from a client

processes the string representing the client request, separating the serialised code in it from the normal arguments in a POST message. The code is unpacked into the heap using the `unpackV` function, and the computation is executed. The computation should generate a new web page with results to be sent back to clients. If there is still the need for more interaction with the client, the web page generated should also contain the continuation of the computation.

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

As the Haskell code is serialised, it is easy to apply encryption, to prevent untrusted parts to access sensitive data or execute the code, and compression to make the messages smaller.

7.2.3 A Counter

As an example we present the implementation of a counter that saves its continuation in the page displayed by the browser, as in Figure 7.72. The counter is implemented as follows:

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

It takes as an argument its current state (an `Int`) and uses the `counterPage` function to generate the response sent back to web clients. The `counterPage` action generates a

```
case (check msg) of
(...)
 Get -> do
    str <- counter 0 emptyArg
    sendResp socket str
    sClose socket
```

Figure 7.74: Receiving code from a client

new web page (a `String`), that displays the current state of the counter in HTML (its first argument), and uses the `packV` function to serialise its second argument, that is, the continuation of the computation. The `counter` just ignores its second argument, the contents of the POST message.

Now, every time that a browser sends a request for the root document (/) the counter is started with zero, as in Figure 7.74, and the page generated by the counter, containing its continuation, is sent back to the client. When the user presses the submit button in the web page (Figure 7.72), and the POST message arrives in the web server, the continuation is unpacked, run, and a new continuation is sent back to the client, as demonstrated in case for `Code` (Figure 7.73).

7.2.4 A Distributed Web Server

The web server might get overloaded with work if it receives many requests from clients asking to execute computations. In a mobile language it is easy to offload a server by sending computations to be executed on other locations. We can use *m*Haskell to make our web server distributed: a cluster of machines can be used to run the computations sent by clients. A *thread farm* can be used in order to provide round-robin scheduling of the tasks in the machines available for processing.

The thread farm is implemented using a server that reads actions from a MChannel, and sends these actions to be executed on remote machines that it gets from another MChannel (Figure 7.75). The `threadFarmServer`, when started, returns two MChannels, used to dynamically increase the number of computations and machines used in the distributed system. After creating the two MChannels the main thread of the server, `serverth`, is forked.

The server thread, reads `Maybe (IO())` values from the `ioc` MChannel. The main thread is stopped once it reads a `Nothing` from `ioc`. When the thread finds a value

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

Figure 7.75: The thread farm server

in the MChannel, it gets one remote machine from `hostc` and starts another thread, `handleC`, to handle the execution of the remote computation. The `handleCon` function starts the remote execution of `action` on `host` and, after it completes, `host` is returned to the MChannel of free machines. Remote evaluation (`reval`) is used in this case, instead of `rfork`, because we want to be sure that the remote machine being used in the computation is only returned to the list of free machines once the remote computation has been completed. In the case of the thread farm, as all computations have type `IO ()`, the value returned by `reval` is just the unit value `()`, indicating that the remote job was executed. MChannels, when used locally, work as a concurrent Haskell Channel: values are written and read as in a FIFO queue.

The `threadFarm` function can be implemented as follows:

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

It takes as an argument a list of actions to be executed on remote locations, and a list of locations. It returns an MChannel that can be used to send more computations to the thread farm, or to stop the thread farm server by writing a `Nothing` in it. The `threadFarm` starts the server and then writes the initial values and locations into their respective channels.

In the case of the web server, the `threadFarm` can be started with an empty list of computations:

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

and the `tfMChannel` returned by `threadFarm` is used to send the computations received from clients to the remote thread server.

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

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

The computation is sent to the thread farm through the `tfMChannel`, and it is wrapped in the `execComp` action, that just executes its argument, and sends its result

back to the web server through a channel. Exactly the same approach is used to implement remote evaluation in terms of `rfork`.

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

7.3 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 to decide when and where to move, even when the host that launched the agent is no longer connected to the network. In this Section we describe how *m*Haskell can be used to implement a simple mobile agent platform, based on the Agent Tcl platform [GKN+96], that supports *partially connected* devices i.e., devices that are not always connected to the network, such as laptops and PDAs. Mobile Agents are an interesting programming paradigm when partially connected devices 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

7.3.1 The Docking System

The mobile agent platform presented here is based on the idea of a *docking system* illustrated in Figure 7.76. 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:

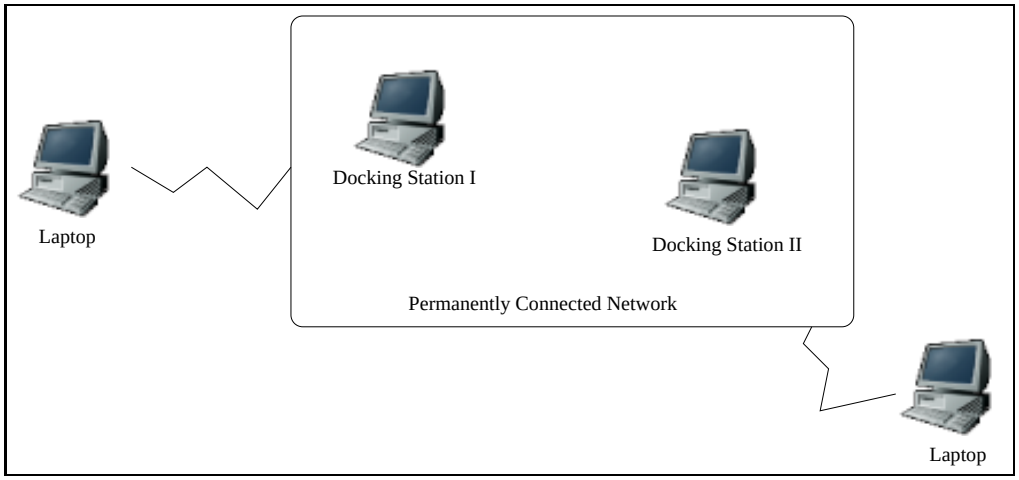


Figure 7.76: Mobile Agent Platform

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

Every time an agent wants to migrate from one location in the network to another, it asks the docking station for permission. The docking station contacts the destination and if it is ready to receive the agent, permission for migration is given and the state of the agent is updated in the docking station. This process is described in Figure 7.77.

```
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
         Located h  -> do
            processMigration returnch newhost statea
            agentLoop statea mch
  (...)
```

Figure 7.77: Processing messages sent by the agent

The docking station has one thread to manage each agent registered (`agentLoop`). When an agent executes the `moveTo` primitive, it 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 an agent is kept inside of an MVar (see Section 2.5.2), 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 (`Located h`), `processMigration` is called (Figure 7.78).

```
processMigration 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 7.78: The `processMigration` function

The `processMigration` 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, it has to send 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 the serialisation primitives `packV` and `unpackV`, 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 it is used, together with the docking station, to send messages to the agents, as described in Section 7.3.4.

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 7.3.4.

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 migrates from the machine that created it, it can only be reached through the docking station using its `MAgentID`.

7.3.2 Finding Resources

As described in Section 4.2, *m*Haskell already provides primitives for resource discovery and registration. All locations running *m*Haskell programs must also run a registration service for resources that is used to register resources with names and to retrieve the resources available. Exactly the same mechanism can be used by agents to find and use resources.

We do not describe here how agents find the names for resources and and where they are located, as it is a well studied problem in many areas, e.g., in data bases and operating systems. Agents could find these names in distributed databases, or *yellow pages* [GKN+96], where resources can be registered and accessed as in a peer-to-peer network.

7.3.3 Communication

An agent should be able to communicate with other agents and with its owner. The docking station could contain a *post-office* for each agent, where messages could be stored, and/or forwarded to the location where the agent is. The primitives for communication only need to know the docking station for the agent, probably using the agent's id, and messages can be sent to the post-office and stored there until the agent wants to read them (as in Figure 7.79).

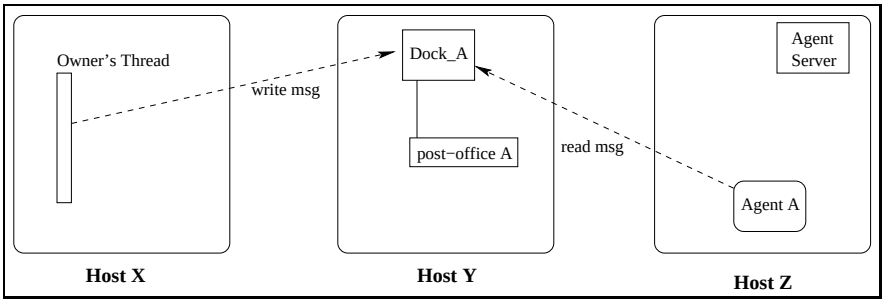


Figure 7.79: The Post-Office

Furthermore, we want the agents to be able to send messages back to their owner whenever they are connected. In Section 4.6.1, we gave an implementation of multiple reader channels, called `PIChannel`s, that are also based on the idea of a post-office: an MChannel is created in a remote location, and threads reading and writing to the `PIChannel` send messages to this location. A similar approach can be used to provide communication in the mobile agents platform by forcing the agents to create the `PIChannel` in the location that contains their docking station.

7.3.4 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:

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

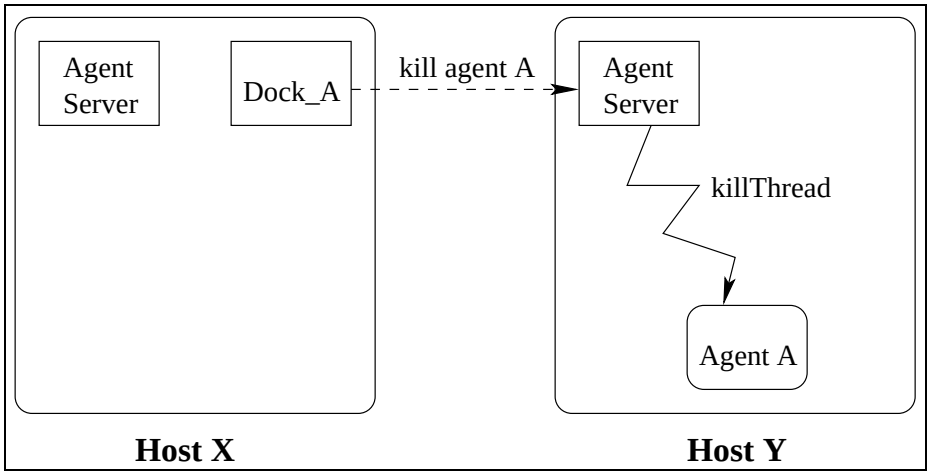


Figure 7.80: Killing a Mobile Agent

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 7.80).

Haskell supports asynchronous exception [MPJMR01], and the same approach used to kill an agent can be used to raise an exception in a remote thread.

7.4 Summary

This Chapter evaluates *m*Haskell for engineering realistic mobile applications, including a distributed meeting planner, a distributed stateless web server and a platform for mobile agents. The abstractions presented in previous Chapters were heavily used. In the distributed meeting planner all the coordination aspects of the application are

hidden in mobility skeletons, and the application essentially only requires parameterisation of the skeletons. The stateless web server relies on the ability of a mobile language to save and restore the application state at arbitrary points during execution. This illustrates how *m*Haskell's state-preservation technology can used for other purposes, e.g., for persistence. The web server was made distributed by using a new mobility skeleton, a thread farm, demonstrating that skeletons similar to algorithmic skeletons can be implemented in a mobile language.

*m*Haskell makes the development of traditional mobile applications easier, as well-tested code can be reused in the form of a mobility skeleton, and the programmer does not have to worry about lower level details of mobility, she can just use a skeleton making the application more reliable and easier to write.

*m*Haskell can express more powerful mobile infrastructures. For example mobile hardware, such as laptops, can benefit from mobile computation as users can initiate mobile programs to perform a task while they are disconnected, and only connect later to collect results. We show an *m*Haskell implementation of a simple mobile agent architecture that supports partially connected computers, providing agents that can communicate, be found and killed, even when the user that created it is not connected. We also describe how to use *m*Haskell to model powerful features such as location independent communication and distributed asynchronous exceptions.

The purpose of the applications presented in this Chapter is demonstrate the use of *m*Haskell and the abstractions for mobility given in the last Chapter. Many issues that are important for real world applications were left out. For example, the meeting planner has no mechanism to ensure that an available time is not given to more than one agent. The objective of the application was only to show that the pattern of mobility present in the meeting planner can be expressed using mobility skeletons. The problem of synchronisation of the time tables should be solved outside the skeletons, i.e., in its arguments, or in the resources used.

Security was also left out in the examples. Authentication could be provided through cryptography: all the mobile code being communicated is serialised into to a string, and this string could be easily encoded using public keys.

Chapter 8

Conclusions and Future Work

This dissertation is about *implicit* and *explicit* mobility in purely functional languages. In *implicit mobile languages*, computations are moved automatically by the runtime system of the language to better use the computational power available. *Explicit mobile languages*, give the programmer control over the placement of active computations and execute on open networks where programs can join and leave the distributed system at any time.

Section 8.1 reviews the contributions of this dissertation; Section 8.2 outlines future work; and Section 8.3 discusses the lessons learned with this work.

8.1 Contributions

The contributions of this dissertation are:

Runtime system support for implicit mobility in a purely functional language. Semi-explicit parallel functional languages usually have automatic mechanisms for the distribution of potential work, i.e., unevaluated expressions. In these systems, it is common for some processors to be idle while others have many runnable threads. The performance of these systems can be improved if besides potential work, threads could also be migrated. GUM (Graph reduction for a Unified Machine model), is the runtime system that underlies the implementation of a number of Haskell extensions for parallel programming (i.e., GpH [THLP98], Eden [BLOMP97] and GdH [PTL00]). We have extended the GUM runtime system with a mechanism for the migration of threads. Migrating a thread incurs significant execution cost and the system implemented uses

a sophisticated mechanism to choose when to migrate threads. Measurements of non-trivial programs on a high-latency cluster architecture show that thread migration can improve the performance of data-parallel and divide-and-conquer programs with low processor utilisation.

A purely functional language for explicit mobility: *m*Haskell is a Haskell extension for writing distributed mobile software. It is the first explicit mobile language based on a purely functional language. It extends Concurrent Haskell with a set of low-level primitives for communication and *mobile channels* (MChannels). MChannels are higher-order, single-reader communication channels that allow the communication of any Haskell value including functions and channels. An operational semantics for the *m*Haskell primitives is given. Programming with MChannels is low-level: the programmer has to specify aspects such as communication and synchronisation of computations, and if mobile values are not carefully managed, e.g., using dynamic types, type errors may occur at runtime. Conventional medium-level abstractions for mobile computation such as *remote thread creation* and *remote evaluation* are readily defined in *m*Haskell, using Haskell's first-class computations. *Strong mobility* is defined as the combination of weak mobility, higher-order channels and first-class continuations.

New high-level mobile coordination abstractions: *Mobility Skeletons* are higher-order polymorphic functions that encapsulate common patterns of mobile computation. Although many mobile languages are based on functional programming languages, as far as we know this is the first time common communication behaviours in mobile programming were specified as higher order functions or skeletons. The range of *m*Haskell abstractions have been used to implement both conventional mobile applications such as a distributed meeting planner and a stateless web-server. We also demonstrated, by implementing a small mobile agents system, that *m*Haskell can be used to model a more sophisticated architecture for mobile programming, supporting powerful features such as location independent communication and distributed asynchronous exceptions.

8.2 Future Work

Following the work initiated in this dissertation, a number of issues could be further investigated:

- *Implicit Mobility*: In future work it may be possible to better characterise programs and architectures where thread migration may be beneficial. GUM's thread migration mechanism and load management policies, described in Section 3.3, could easily be improved, e.g. replacing the random targeting of FISH and SHARK messages with a more focused approach; and possibly recording partial load information in all messages to maintain a time-stamped partial load information on each PE.

- *Enhanced Type Systems*: Haskell is a statically typed functional language and there is a lot of research on using static-types to enforce safety and security in mobile languages [AGH+04, Tho97]. In [Kir01] Zeliha Kirh describes a static type system for an extended λ-calculus with communication channels, similar to MChannels, which statically predicts which values in the program might be transmitted to remote locations. *m*Haskell's type system could be extended with such analyses that would identify the parts of the program that should be compiled into byte-code and the parts that can safely be compiled into machine code.

- *Reasoning about mobile programs*: We are still in the process of developing a full semantics for *m*Haskell to use it for proving properties of evaluation location and evaluation order in mobile programs. In particular, it would be useful to specify and prove the behaviour of the mobility skeletons. For such proofs the transitional level of the semantics is the most important one, and starting from the existing semantics for Concurrent Haskell [PJ01] we managed to restrict our extensions to only this level (Section 5.2). The only exception is the `forceThunk` function, which we still have to formalise. Another area of future work would be cost models for our mobility skeletons to predict when and where computations should migrate. Identities like `mmap_ (f>>g) hs = (mmap_ f hs) >> (mmap_ g hs)`, that can be proved by induction over the list of hosts, are very useful to prove properties about mobile programs, and could be further investigated.

- *Strong Mobility*: Programs using strong mobility might look awkward if most of the actions executed in the mobile thread belong to the IO monad, as IO actions have to be lifted into the continuation monad (see Section 6.3.1). 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 ()`.

- *Distributed Asynchronous Exceptions*: The same method used to kill remote agents in the mobile agent system described in Section 7.3.4, could also be used to raise exceptions in remote computations. The mobile agent system presented in in this dissertation can serve as a model for implementing a library for distributed asynchronous exceptions in Haskell.

- *Security and Safety*: In a system where computations move between locations *safety* (received computation should not be the cause of runtime errors that may prevent the program to present its expected behaviour) and *security* (protection against malicious code) are important issues. This dissertation focuses mainly in how to express mobility of computations in a purely functional language and we believe that most of the issues related to safety and security should not appear in the language but be provided by its implementation. *m*Haskell's implementation could easily be extended to provide security through authentication: all the mobile code being communicated is serialised into strings, and these strings could be easily encoded through cryptography, e.g., using the public key of the sender/receiver. The current implementation could also be extended to provide sandboxing: the runtime system receiving mobile code from an *untrusted* source, could provide safer implementations for *dangerous* functions (e.g., `unsafeperformIO`), and the alternative implementations are linked to the received code. This *safer* implementation of a function could just raise an exception saying that this function is not allowed. The security and safety issues of dynamically linking mobile code have been much studied, e.g., [Sew01, SLW+04], and although not currently implemented, some of these techniques could be incorporated into *m*Haskell.

- *Mobility Skeletons*: The idea of mobility skeletons could be applied in other programming paradigms. Design patterns encapsulate solutions to recurrent problems in object oriented software design. A pattern describes a common problem

and provides the essence of the solution to this problem, often as a collection of classes. To facilitate their use, patterns are classified and catalogued, as for example in [GHJV95]. In appendix A, we give an implementation of the `mmap_` skeleton from Section 6.2.1 as a *template design pattern* using Voyager [Voy06], which is a Java IDE supporting code mobility.

Another obvious direction for future work would be to analyse commercial and research mobile applications trying to identify more mobile coordination abstractions that are general i.e., applicable in many cases, realistic i.e., useful for real applications, and easy to reason about.

8.3 Discussion

Previous work on mobile computation suggests that the strong expressive power provided by higher-order functions in functional languages is also an important abstraction for mobile languages [Kna95, Kir01]. Hence many mobile languages are based on functional languages. In this thesis, we have demonstrated that from a very small set of stateful communication primitives and higher-order channels, higher level abstractions for mobile computation can be built in a functional language with higher-order functions and support for monadic programming. The support for monadic programming in Haskell was very important in the development of *m*Haskell's abstractions. Stateful computations, or monadic IO values, are first class objects of the language, hence they can be *manipulated* in the purely functional part of the language to generate new control structures and abstractions. The clean semantics available for IO actions was also helpful in the development of a simple semantics for the MChannel primitives. An important Haskell feature that does not suit a mobile language is lazy evaluation. Although some interesting programs can be implemented using lazy evaluation, like the alternative implementation of `mfold` using strong mobility described in Section 6.3.5, it is very difficult to predict the amount of data being communicated in a lazy language, as described in Section 5.1.4. The solution to the problem was to force the evaluation of pure expressions before communication. Even though this evaluation usually does not affect the semantics of mobile programs, as those are usually stateful computations that are not affected by this evaluation step, it looks odd in a language like Haskell and makes the semantics of the language more difficult to define.

To make the code presented in this dissertation more readable, we tried to avoid the use of *dynamic types* in the examples when possible. But while building the mobile applications we noticed that they are very important in real-world applications to avoid errors during run time. The current Haskell implementations only provide very limited support for dynamic types, and a commercial implementation of a mobile language similar to the one presented here should provide better dynamic types, probably like the one available in the functional language Clean [Pil98].

Some parallel languages are based on purely functional languages because in such languages expressions can be evaluated at any order without affecting the result of the computation. This characteristic also means that, in an implicit mobile language, running threads can be reallocated to better utilise the processors. Another important lesson in the work presented here was that automatic thread migration, if carefully scheduled, can improve the performance of some parallel programs with low processor utilisation, and can also correct a poor initial load balancing of tasks.

It is expected that the work presented here can contribute in the understanding of the field and may help in the development of new commercial technology for distributed mobile programming.

Appendix A

`mmap_` as a Template Design Pattern

The idea of a template method pattern is to define the skeleton of an algorithm in an operation deferring some steps to subclasses. The template method lets subclasses redefine some steps of the algorithm without changing the structure of the algorithm. In figure A.81 we give an implementation of `mmap_` as a template design pattern. The `Mmap_` class encapsulates a broadcast communication pattern. The method `sendObjects()` is the template method that contains some code written in Voyager [Voy06], a Java IDE supporting code mobility, to send the computations to be executed on remote hosts. This method is defined in terms of two abstract methods, `initObject` and `executeObject`, that should describe how to initiate and execute the object that is going to be broadcasted. The class can be extended to map `Ball` objects as in figure A.82 and used as in figure A.83.

Note that the Java class in Figure A.81 is approximately 30 times more verbose than the mobility skeleton given in Chapter 6.

```
abstract class Mmap_{

  protected Object action;
  protected String[] hosts;
  protected Object proxy;

  public Mmap_ (Object obj,
                String[] hosts) {
    this.action = obj;
    this.hosts = hosts;
    this.proxy = initObject();
  }

  public void sendObjects () {
    int i;
    Object proxy;
    /* ... Start Voyager,
           handle exceptions */

    for(i=0;i<hosts.length;i++) {
      Mobility.of(this.proxy).moveTo(hosts[i]);
      executeObject(); }
  }

  public abstract void executeObject();
  public abstract Object initObject();
}
```

Figure A.81: mmap_ as a Template Design Pattern

```
public class MmapBall extends Mmap_{

  public MmapBall
           (Ball obj, String[] hosts) {
    super(obj,hosts);}

  public void executeObject() {
    ((IBall)this.proxy).hit();}

  public Object initObject() {
    return
      ((IBall) Proxy.of(this.action)); }
}
```

Figure A.82: Extending the Mmap_ Class

```
public class Hello{
  public static void main (String[] args) {
    String[] s = new String[2];
    s[0] = "//lxtrinder:8000";
    s[1] = "//linux33:9000";

    Ball b = new Ball();
    MmapBall mb = new MmapBall(b,s);
    mb.sendObjects(); }
}
```

Figure A.83: Using the Mmap_ Class

Bibliography

[AGH+04] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04 — Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 1–26, Marseille, France, March 10–13, 2004. Springer-Verlag.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of errors.* PhD thesis, Royal Institute of Technology, Stockholm, 2003.

[Bar84] H.P. Barendregt. *The Lambda Calculus.* North-Holland, 1984.

[BHK+94] T. Bülck, A. Held, W. Kluge, S. Pantke, C. Rathsack, S-B. Scholz, and R. Schröder. Experience with the Implementation of a Concurrent Graph Reduction System on an nCUBE/2 Platform. In *CONPAR'94 — Conf. on Parallel and Vector Processing*, LNCS 854, pages 497–508, 1994.

[BJL+95] R.D. Blumofe, C.F. Joerg, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP'95 — Symp. on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, USA, 1995. ACM.

[BL98] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[BLOMP97] 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.

[BN01] L. Bettini and R. De Nicola. Translating strong mobility into weak mobility. In *Proc. of 5th IEEE Int. Conf. on Mobile Agents (MA)*, LNCS 2240. Springer-Verlag, 2001.

[BRS+85] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. Mach-1: An operating environment for large-scale multiprocessor applications. *IEEE Software*, 2(4):65–67, July 1985.

[CAL+89] J.S. Chase, F.G Amador, E.D. Lazowska, H.M Levy, and R.J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Symp. on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, USA, 1989.

[Car97] Luca Cardelli. Mobile Computations. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume LNCS 1222, pages 3–6. Springer-Verlag: Heidelberg, Germany, 1997.

[Car99] 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. IOS Press.

[Car01] Luca Cardelli. Abstractions for mobile computation. In *Secure Internet Programming*, LNCS 1603, pages 51–94. Springer-Verlag, 2001.

[CF99] 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)*, pages 22–29, Palm Springs, CA, USA, 1999. IEEE Computer Society.

[CGK98] Manuel M. T. Chakravarty, Yike Guo, and Martin Kohler. Distributed Haskell: Goffin on the Internet. In *Fuji International Symposium on Functional and Logic Programming*, pages 80–97. World Scientific, 1998.

[CGSv93] D.E. Culler, S.C. Goldstein, K.E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. 18:347–370, June 1993.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5):704–739, 1995.

[Cla99] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, 1989.

[CPV97] A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proceedings of the 19^{th} International Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.

[CT97] Benjamin C.Pierce and David N. Turner. Pict: A programming language based on the pi calculus. Technical report, Computer Science Department, Indiana University, 1997.

[DBLT03] André Rauber Du Bois, Hans-Wolfgang Loidl, and Phil Trinder. Thread migration in a parallel graph reducer. In *IFL 2002*, LNCS 2670, pages 199–214. Springer-Verlag, 2003.

[DBPLT02] André Rauber Du Bois, Robert Pointon, Hans-Wolfgang Loidl, and Phil Trinder. Implementing declarative parallel bottom-avoiding choice. In *Proc. 14th Symposium on Computer Architecture and High Performance Computing*, pages 82–89, Victoria, Brazil, 2002. IEEE Press.

[DBTL03] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Towards a Mobile Haskell. In *Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pages 102–116, Valencia (Spain), 2003.

[DBTL04a] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Implementing Mobile Haskell. In *Trends in Functional Programming*, volume 4, pages 79–94. Intellect, 2004.

[DBTL04b] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Towards Mobility Skeletons. In *CMPP'04 — Constructive Methods for Parallel Programming*, pages 77–93, Stirling, Scotland, June 2004.

[DBTL05a] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. *m*Haskell: mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

[DBTL05b] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Strong mobility Mobile Haskell. In *Draft proceedings of IFL 2005*, September 2005.

[DBTL05c] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288, 2005.

[Erl06] Erlang. WWW page, http://www.erlang.org/, 2006.

[FF95] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer, 4th edition.* MIT Press, 1995.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Conference on Lisp and Functional Programming (LFP'84)*, Austin, Texas, 1996. ACM Press.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, LNCS 1119, pages 406–421. Springer-Verlag, 1996.

[FH01] Volker Stolz Frank Huch. Distributed programming in Haskell: From ports to streams. Unpublished Draft, 2001.

[FK99] Ian Foster and Carl Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[Fla99] D. Flanagan. *Java in a Nutshell, Third Edition.* O'Reilly & Associates, 1999.

[FPV98] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongerra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine.* MIT, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[GKN+96] Robert S. Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report TR96-285, Dartmouth College, May 1996.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT, second edition, 1999.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrend and functional programming. *Journal of Parallel Programming*, 2(18):121–160, April 1989.

[Gro01] William Grosso. *Java RMI.* O'Reilly, 2001.

[GW99] James R. Groff and Paul N. Weinberg. *SQL, The Complete Reference.* Osborne McGraw-Hill, 1999.

[Hal97] David Alan Halls. *Applying Mobile Code to Distributed Systems.* PhD thesis, Computer Laboratory, University of Cambridge, 1997.

[Han99] 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)*, LNCS 1702, pages 376–395. Springer-Verlag, 1999.

[Hoa85] C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP90] Kevin Hammond and Simon Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer. In *IFL'90 — Intl. Workshop on the Parallel Implementation of Functional Languages*, pages 51–72, Nijmegen, The Netherlands, Jun 1990.

[HP92] Kevin Hammond and Simon Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *IFL'92 — Intl. . Workshop on the Parallel Implementation of Functional Languages*, pages 73–98, RWTH Aachen, Germany, September 1992.

[Hue97] Gerard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[ISS98] A. Itzkovitz, A. Schuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *J. of Systems and Software*, 42(1):71–87, 1998.

[JH93] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Technical Report YALEU/DCS/RR-982, Yale University, August 1993.

[Kes96] M. H. G. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, Wiskunde en Informatica, Katholieke Universiteit van Nijmegen, The Netherlands, 1996.

[Kir01] Zeliha Dilsun Kirli. *Mobile Computation with Functions.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2001.

[KLB91] H. Kingdon, D.R. Lester, and G. Burn. The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network. 34(4):290–301, 1991.

[Kna95] Frederick Colville Knabe. *Language Support for Mobile Agents.* PhD thesis, School of Computer Science, Carnegie mellon University, 1995.

[KOMP98] Ulrike Klusik, Yolanda Ortega-Mallen, and Ricardo Pena. Implementing Eden - or: Dreams become reality. In *Implementation of Functional Languages*, LNCS 1595, pages 103–119. Springer-Verlag, 1998.

[LFA96] D.K. Lowenthal, V.W. Freeh, and G.R. Andrews. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing. *J. of Parallel and Distributed Computing*, 37:42–54, 1996.

[LH96] H-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96 — Intl. Workshop on the Implementation of Functional Languages*, LNCS 1268, pages 184–199, Bonn/Bad-Godesberg, Germany, September 1996. Springer-Verlag.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[LO99] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 3(42):88–89, March 1999.

[Loi98] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, University of Glasgow, March 1998.

[Lov93] David B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42, 1993.

[LPJ95] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp Symb. Comput.*, 8(4):293–341, 1995.

[LTB01] H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4):471–486, 2001. Selected papers from HLPP'01 — International Workshop on High-level Parallel Programming and Applications, Orleans, France, 26-27 March, 2001.

[LTH+99] H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1999.

[Mar00] Simon Marlow. Writing high-performance server applications in Haskell, case study: A Haskell web server. In *Haskell Workshop*, Montreal, Canada, September 2000.

[Mar05] Simon Marlow. The Glasgow Haskell Compiler. WWW page, http://www.haskell.org/ghc, 2005.

[MDW99] D. Milojíćić, Frederick Douglis, and Richar Weeler. *Mobility: Processes, Computers, and Agents.* Addison-Wesley, Reading, MA, USA, 1999.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil93] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, May 1999.

[MLPJ99] A. K. Moran, S. B. Lassen, and S. L. Peyton Jones. Imprecise exceptions, co-inductively. In *Proceedings of HOOTS'99*, volume 26 of *ENTCS*, 1999.

[MMS95] B. Mathiske, F. Matthes, and J.W. Schmidt. On Migrating Threads. In *Intl. Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel, June 1995.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press, 1989.

[MPJMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in Haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[MR96] E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software — Practice and Experience*, 26(3):327–356, March 1996.

[New06] Jeff Newbern. All about Monads. WWW page, http://www.nomaware.com/monads/, 2006.

[Nik95] R.S. Nikhil. Parallel Symbolic Computing in Cid. In *Workshop on Parallel Symbolic Computing*, LNCS 1068, pages 217–242, Beaune, France, Oct. 1995. Springer.

[NS94] R.S. Nikhil and A. Singla. Automatic Granularity Control and Load-Balancing in Cid. Technical report, DEC Research Labs, December 1994.

[NSvEP91] 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)*, LNCS 505, pages 202–219. Springer-Verlag, 1991.

[OCa06] OCaml. WWW page, http://caml.inria.fr/ocaml/, 2006.

[OCD+88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ACM CR 8905-0314*, 21(2), 1988.

[Pil98] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, LNCS 1595, pages 169–185, 1998.

[PJ92a] 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.

[PJ92b] S.L. Peyton Jones. *Implementation of Functional Programming Languages. A Tutorial.* Prentice Hall, 1992.

[PJ01] 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.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*. ACM Press, Jan 93.

[PS02] R. Peña and C. Segura. A polynomial cost non-determinism analysis. In *Implementation of Functional Languages*, LNCS 2312, pages 121–137. Springer-Verlag, 2002.

[PTL00] R. Pointon, P.W. Trinder, and H-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *IFL 2000*, LNCS 2011, pages 53–70. Springer-Verlag, 2000.

[RBMS97] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace Conference*, pages 79–91, 1997.

[San01] Davide Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253(2):311–350, 2001.

[Sek99] Tatsurou Sekiguchi. *A Study on Mobile Language Systems.* PhD thesis, Department of Information Science, The University of Tokio, 1999.

[Sew98] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. of the International Colloquium on Automata, Languages and Programming*, LNCS 1443, pages 695–706. Springer-Verlag, 1998.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (London)*, pages 236–247, January 2001.

[SJ95] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *15th ACM Symp. on Operating Systems Principles*, pages 68–77, 1995.

[SLW+04] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. design rationale and language definition. Technical Report UCAM-CL-TR-605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp.

[Sri95] R. Srinivasan. Rpc: Remote procedure call protocoll specification version 2. Technical report, Sun Microsystems, 1995.

[TA03] A. Tolmach and S. Antoy. A monadic semantics for core Curry. In *Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pages 33–46, Valencia (Spain), 2003. Universidade Politecnica de Valencia.

[THLP98] 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.

[THM+96] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.

[Tho97] Tommy Thorn. Programming languages for mobile code. *ACM Comput. Surv.*, 29(3):213–239, 1997.

[TLP02] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(14-15):469–510, 2002.

[TV96] J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon '96*, pages 58–63, 1996.

[TW85] Ed Taft and Jeff Walden. *PostScript Language Reference Manual.* Addison-Wesley, 1985.

[Uny01] Asis Unyapoth. *Nomadic π-Calculi: Expressing and Verifying Communication Infrastructure for Mobile Computation.* PhD thesis, Pembroke College, University of Cambridge, 2001.

[vCGS92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *ISCA'92 — Intl. Symp. on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992. ACM Press.

[VF01] V.Stolz and F.Huch. Implementation of Port-based Distributed Haskell. In Thomas Arts and Markus Mohnen, editors, *Draft. Proc. of IFL 2001*, 2001.

[Vol96] Dennis Volpano. Provably secure programming languages for remote evaluation. *ACM Computing Surveys*, 28(4es):176–176, 1996.

[Voy06] Voyager System. http://www.recursionsw.com/voyager.htm, 2006.

[vWP02] Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In *IFL 2002*, LNCS 2670, pages 215–231. Springer-Verlag, 2002.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM Press, 1990.

[Wad95] Philip Wadler. Monads for functional programming. In E. Meijer J. Jeuring, editor, *First International Spring School on Advanced Functional Programming Techniques*, LNCS 925, pages 24–52. Springer-Verlag, 1995.

[Weg71] P. Wegner. *Programming Languages, Information Structures and Machine Organisation.* McGraw-Hill, New York, 1971.

[Woj00] Pawel Tomasz Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Wolfson College, University of Cambridge, 2000.

[ZTML05] A Al Zain, P. W. Trinder, G. J. Michaelson, and H-W. Loidl. Managing hetrogenity in a grid parallel haskell. In *Intl. Conference on Computer Science (ICCS'05)*, LNCS 3514-3516. Springer-Verlag, 2005.