Implementation of a language with strong mobility

Stephane Epardaud

Université de Nice - Sophia Antipolis **UFR Sciences**
École Doctorale Sciences et Technologies de l'Information et de la Communication

THÈSE

Présentée pour obtenir le titre de

Docteur en Sciences de l'Université de Nice - Sophia Antipolis

Spécialité Informatique

par
Stéphane ÉPARDAUD

Mise en œuvre d'un langage à mobilité forte

Thèse dirigée par Manuel SERRANO et Gérard Boudol
et préparée à l'INRIA Sophia, projet MIMOSA

Soutenue le 18 Février 2008 devant le jury composé de :

MM.	Michel RIVEILL	Président	UNSA
	Shriram KRISHNAMURTHI	Rapporteur	CS-BROWN
	Jean-Bernard STEFANI	Rapporteur	INRIA
	Manuel SERRANO	Directeur de Thèse	INRIA
	Gérard BOUDOL	Directeur de Thèse	INRIA
	Marc LACOSTE	Examinateur	FT R&D

Thanks

I love deadlines. I like the whooshing sound they make as they fly by.

– Douglas Adams

Many thanks to all the people who helped me during this thesis, especially my family and the MIMOSA project. Not many thanks to my phone: my one and only embedded test platform, how could you let me down?
Many grateful thank yous to the Merriam-Webster online [MW]: every language should be documented (partly by dictionaries) and that documentation should be accessible for free for everyone wishing to speak, read or write it. May all languages see the light and make their grammar, vocabulary and other documentation available for free on paper or online.

Contents

Chapter 1

Introduction

Getting an education was a bit like a communicable sexual disease. It made you unsuitable for a lot of jobs and then you had the urge to pass it on.

– Terry Pratchett

This report, by its very length, defends itself against the risk of being read.

– Winston Churchill

Although computers have been around and connected for some times now, these last few years have seen an explosion in the number of computing platforms available in many homes. Not only is it typical for a household to possess several computers (sometimes as many as a PC and a laptop per person), nowadays many home appliances and small devices have become what we could describe as computing platforms, short of calling them full-fledged computers. From modems to routers, used to connect the several traditional computers, but also from game consoles and digital video recorders (DVRs) to mobile phones, which are becoming more and more open as well as powerful.
All these devices which used to work on very closed and custom proprietary hardware and software, now take the direction of using standard operating systems and become open to the development of third-party software. The number of devices at home where a user can (and will) install new software has grown substantially. In many cases the user even expects similar programs to run on totally different hardware. See for example web applications which are expected to host word processors or agendas with the same interface on a computer, a mobile phone, or on game consoles (portable or not).
The example of web applications is especially illustrative of some of the problems raised by such miscellaneous computing platforms. Due to their extremely varying hardware, operating systems, software libraries, development and deployment methods, it is easier to write web applications because most of these devices feature a web browser. This common denominator allows web software developers to deploy applications written for standard computers on any of these new devices (phones, DVRs) with minimal new development based on the promise that web applications use features standard across every web browser. In order to take advantage of features offered by the device outside of those offered by the web browser, one has to write a custom application for the given device. For instance in order to control the routing tables on a router, the sending of messages on a mobile phone, the recording schedule of a VCR or the current playlist of a media center, one has to write

custom software on each of these devices. Each software will not only deal with different libraries (controlling a router takes a different API than a media center) but also different operating systems, programming languages, distribution and packaging means. There are however efforts to standardise the development on these devices, for example by using a Java platform such as J2ME on devices where the full-fledged J2SE platform is not (yet) available.

If one were to need interaction between all these applications, on the other hand, one would have to convert each of these applications into client/server applications, and presumably have a central application to control everything. This is still the accepted pattern to this day, and may explain the fact that these devices interact so little. But we can see many examples of why it would be desirable to make these devices interact. When the network goes down on the modem, the phone could send an SMS (short message system) to the house occupants if they are not home at the time (this detection can be as simply as checking if their mobile phones are in the house's network). At the desired wake up time, the home server could turn the laptop on, instruct the media player to play a nice song, then display the list of shows recoded during the night on the laptop. If only writing such trivial (in the sense that its expected behaviour is trivial) applications could be done as easily as writing the small scripts written regularly to automate tasks to perform on standard computers...

We think that *mobility* is a good solution to these problems, and in particular a valid alternative to the client/server solution. We would like to be able to write a simple program which waits for a certain time before it *migrates* to the media player to instruct it to play a song using the standard API available on this device, then migrate again on the DVR to retrieve the list of recorded shows during the night, before it migrates again on the laptop after it has been turned on, in order to display this list. The laptop user could then click on some of these shows to instruct our mobile program, or *agent*, to transfer the videos on the mobile phone or portable media player for later viewing during train commuting.

While all the above could be accomplished using a client/server approach, it would require more work. Indeed, on each device we would need to design a server to handle incoming requests for the local operations, then a library for the client so that it can invoke those services on the server. While there are standard protocols to implement the required communications between the client and the server, the fact that the list of operations has to be defined on the server makes it much less flexible than by using migration. Because an agent can come with its own code on the server, it can accomplish a much richer (if not infinite) set of tasks than the limited operations made available by the server. Furthermore, if some parts of our network would be protected by firewalls, remote services would fail or have to be forwarded correctly by the firewall. With agents on the other hand, once the agent would be authorised by the firewall, it could freely roam to the protected devices to accomplish its task.

At the same time, in the industry, large numbers of networked computers have become not only common, but standard. It is widely accepted that, with the proper software and hardware architecture, scaling is done by mostly throwing hardware at the problem. This is, of course, a derisory simplification on our part, but most very-large-scale web applications are known for the enormous network of standard computers they require. In such contexts, a lot of time and effort is spent in finding how to divide a computing task between several computers, and how they will communicate to achieve these different parts.

Again we think that mobility is an adequate solution for such distributed applications. For instance, upon arrival of a new request, a agent could start on a dispatching server, then

migrate to a data source server to gather the data needed for the response. The agent would then migrate to a computing server to work on the gathered data before presenting it to the user.
Exploiting the locality of resources and load distribution are only examples of the use of agent mobility. There are many other areas in which agents can be used. But programming with agents is very different from client/server applications and poses its own problems, especially related to agent communication and their scheduling. We believe that parallelism, be it on different computers, or of several agents on the same computer, is a model that is, at the same time, desired and feared by programmers. It is well known that access and modification of shared data between parallel programs (again, on one or several computers) leads to data races and memory corruption unless the appropriate protections are taken by every parallel program.
We believe programmers should be given some control over parallelism when it is possible (on one computer), and be prevented from accessing shared data when it is not possible to control parallelism (between separate computers). We also believe this requires special support from the programing language itself, in order to ease the creation, migration and scheduling of agents for the programmer.
To this end, we present ULM (Un Langage pour la Mobilité): a language with agent mobility, aimed at easing the creation and mobility of agents, as well as offering logical semantics to their local scheduling, and data isolation across the network.
In the second chapter, we present the state of the art on agent mobility. The language and its features are presented in the third chapter, while the implementation of the language is described in chapters four and five. The sixth chapter concentrates on the interfacing between ULM and other languages, and chapter seven focuses on the use of ULM as an alternative to event loops. In chapter eight we present some applications of ULM. We finally present future directions and conclude in chapter nine.

Chapter 2

State of the art

Je ne suis ni pour, ni contre, bien au contraire.

– Coluche

Agent mobility has been the subject of much research [FPV98] and many implementations of various mobility flavours have materialised. In this chapter we are going to introduce the many subtleties of agent mobility.

2.1 The various forms of mobility

There are several forms of program mobility. When attempting to move a program from one computer to another, the oldest and most obvious means is to copy the program from one computer to another and start executing the program on the new location. This is the crudest means of program mobility: where the program's user has to consciously take physical action to move one program from a location to another.
It is highly commonplace for computers to move programs automatically nowadays. For example, most program installers nowadays automatically download the latest version of the program they are installing from the internet. Even on web pages there are programs that are downloaded from the web server to the web client, which then get executed on the client. Even though the user is not conscious, or aware of this mobility, it is still happening.
Although the act of loading a program from a remote location can be seen as a form of mobility, it can only be seen as a form of *code* mobility. In the cases described above, the user (or a program) initiates the transfer of a program's *code* and then proceeds to start the transferred program. What is more commonly referred to as *program mobility* is the decision of one program to move *itself* (or another *running* program) from one location to another.
The distinction, more precisely, is that the program is being moved while it is running: it does not need to be restarted on the new location. Usually when talking about mobility it is meant that not only code is moved, but also the state of the running program (whether automatically or not).
There are two types of mobility for running programs. *Weak* mobility refers to moving a running program without its call stack (or continuation). For example, the program would need to specify which function should be called after moving, with no possibility of staying within the current function during migration. The ability to move a running program with

its call stack (thus, being able to move in the middle of any computation) is called *strong* mobility.
Traditionally, mobile programs are called *agents*, and the different locations on which they can move are called *sites*, while the act of moving is actually called *migrating*.

2.2 Uses of mobility

Mobility has often been considered for widely differing solutions:

Distribution
In order to split the computational cost of a program, it is often divided into smaller units of computation which can executed in parallel on different processors, often on different computers. In this case mobility would allow such smaller units to move to an idle computer to perform the computation.

Physical movement
Using mobility, a running program, such as a game or a mail reader, could move from one computer to another in order to follow the physical movement of its user. This can also be used in order to keep a program running while the hardware it is using is being upgrade: by moving temporarily the program until the hardware has been replaced.

Localised resources
Sometimes a resource can only be obtained from a location. For example in order to use a device physically attached to one computer. A mobile agent could move to a mobile phone in order to send a text message.

Resource proximity
When remote programs need to interact by communication, it may make sense to use mobility to bring those programs on the same site for the duration of their exchange.

Deployment / resource management
Agents can be used for automatic deployment of programs on many sites. They can also be used in order to manage resources on distant computers. For example an agent can be used to locate a software component on one site, then move it to another site to install and configure it.

Some of these properties can of course be combined: in Second Life [Lif] (an online game) agents are used to represent persons or objects in a virtual world, while sites are used to represent geographical locations of that virtual world. Those sites are then located in different regions of the real world, so these agents are used for distribution, physical movement and resource proximity.
Erlang [VWW96] is famously used in telephony systems to handle dynamic upgrades to the systems without any downtime, by using weak migration to distribute the new code and manage the systems. Automatic reconfiguration and monitoring of complex network systems is also an area where agents can be found [TKT+07].

2.3 Differences between agent systems

Each agent system has its own unique set of features and solutions. They differ in several key areas such as their language, scheduling, memory model, communication and data handling. We will attempt to describe the various differences in each areas.

2.3.1 Language

Some languages have been extended in order to support agent mobility (Kali Scheme [CJK95], Termite Scheme [Ger06], Aglets [LOKK97], Jocaml [CF99]), while other languages have been created especially for mobility (Erlang, LSL, Obliq [Car95] [Car94]). Introducing mobility in an existing language can be a hard task because mobility can have an impact on many of the areas we describe here, including fundamental ones such as scheduling or the memory model.
Because each language has different features, languages with support for continuations can easily implement strong migration with only the proper communication features. Indeed, if a program is able to copy a continuation from one site to another and then invoke it, it can be seen as a form of strong migration (and for all intents and purposes it is strong migration). This is indeed the approach taken by Termite Scheme, Kali Scheme and others [Sum00]. Stackless Python's tasklets can also be called strong migration agents, in the sense that their continuation can be serialised, and thus transmitted by the network to be resumed remotely.
Mobility languages based on Java usually lack strong migration, but there are various efforts to implement continuations in Java like Javaflow [Com] or Brakes [COR], or using a special Java Virtual Machine (JVM) [AAB+05] to implement strong migration for Aglets [CFLQ06].

2.3.2 Memory models

In some models, agents are isolated from one another by having separate address space, like processes. This is the case for Erlang, Termite and LSL. Other agents are more similar to threads in that they share the memory address space, such as Aglets, Kali Scheme and Obliq. The difference in memory model leads to differences in communication and synchronisation between the agents which we will discuss below.

2.3.3 Scheduling

Just as there are several memory models, there are various types of scheduling for agents. Agents with a separate memory model are usually scheduled asynchronously like processes. Most shared memory agents are also scheduled asynchronously like threads (Aglets for example). In the case of LSL, agents are event-oriented, and resemble automata, but it is not clear how these agents are scheduled. Stackless Python's tasklets can be scheduled cooperatively or preemptively depending on the programmer's choice.

2.3.4 Communication

Ideally, agents should be able to communicate together in order to form more complex behaviour than possible with single agents. This communication usually involves synchronisation between the agents, so we will talk about these two aspects at the same time.

Mobile agents with a shared memory model usually use the shared memory for both synchronisation and communication. For example in the case of asynchronous agents such as Obliq and Kali Scheme, locks and condition variables are used for synchronisation between agents while communication is done through shared memory.
Mobile agents without a shared memory model usually resort to *channels* or *mailboxes* to communicate values and synchronise their scheduling. Jocaml [CF99], an extension to Objective Caml [Cam] inspired by the Join-Calculus [FG96], uses channels for communication and synchronisation. In Jocaml, a channel is a bi-directional network-transparent pipe created by agents or processes in order to listen to incoming messages and respond to them. An agent can then locate a given channel in a *naming service* where channels are registered, and communicate with the agent or process on the other end of the channel. A channel is a many-to-one communication means. Stackless Python also uses synchronous channels for its tasklets' communications, but because there can be several listeners, it is a many-to-many communication means.
Erlang, Termite Scheme and Aglets use mailboxes to communicate between agents. Arguably, mailboxes are very similar to channels, with a few differences. In Erlang and Termite Scheme for example, each agent is assigned a globally unique Pid (process identifier) which can be used to send a message to the agent. Each message is sent asynchronously, and queued in a mailbox until the agent decides to read the messages. The agent can decide to read the messages at any point in its execution, and in any order, with the possibility of selecting the message it wants to read first. By communicating Pids in the message, it is possible to obtain synchronous communications between agents.

2.3.5 Data handling

When an agent migrated from one site to another, whether it is strong or weak migration, usually the data used by the agent is affected by migration in one way or another. The simplistic approach is to say that all the data that an agent needs to work on will be migrated along with the agent so it can keep on working on it. This is a simplistic view which poses many problems and the reality is that it is often much more fine-grained.
Agents without a shared memory model usually can migrate with all their data: since it cannot be used by any other agent, it makes no sense to leave it behind. With shared memory it becomes much harder to decide the best way to keep sharing data. If the same data is shared by two agents on one site, and one agent migrates away, suppose the data is copied during migration, if the two agents modify the data and then migrate back on the same site, what sort of synchronisation can we expect from the two different instances of data which used to be shared? This question has no perfect universal solution, and there are many ways to handle the problem.
Obliq, for example, states that free variables (those defined outside of the agent) are transformed into transparent network references upon migration. This means that two remote agents would indeed keep on accessing a unique value across the network.
Sometimes it is not sufficient to know what should happen of variables or references after migration, because it is possible to have the same instance of a data stored into different variables or references of differing types. In some cases we also have to know what happens to values, as opposed to variables or references.
In λdist [SY97] for example, it is possible to associate very fine-grained migration behaviour on values themselves. In this system, a value can be marked to be copied during migration, to always stay on the same location and be accessed remotely after migration, to move along

with the agent and leave a remote reference on the departing site, to become undefined after migration in the agent, to go with the agent and become undefined on the departing site, or to be dynamically rebound after migration. This last example is called *ubiquity*: the ability for a value (or a reference) to have a similar (or not) value on each site, which will always be dynamically rebound after migration.
Kali Scheme has yet another view on the subject and proposes a notion of *address space*, in which objects are allocated, and a special notion of proxies to these objects. A proxy object points to a value in a given address space, but this value can be affected and read in any other address space, without synchronisation. With this type of proxy, it is not possible to remotely affect a value, one has to move to the proxied value's address space to do so. On the other hand, it enables a sort of ubiquity to remote values.

2.3.6 Security

Obviously, mobility poses security questions. Simply allowing any incoming agent to execute any code can be a security problem. There are various possibilities to treat this problem: it is possible to trust some agents depending on its origin (within a trusted network for example), or based on some trusted signature of its code. It is possible to have a finer-grained security as well: by limiting certain operations for agents with insufficient privileges, or limiting their execution time.

2.4 Why we would need yet another mobile agent language

Many systems offer memory barriers across agents, whether they are located on the same site or on different sites. While this ensures that there cannot be asynchronous modification to shared data, notorious for causing problems, we believe the cost of not sharing memory within the same site is too high to justify this solution. Indeed we would like our agents to be able to communicate using shared memory on the same site, in a manner which is free from asynchronous interferences.
Those systems which allow shared memory between agents often allow network references, which is just another asynchronous way to share memory across sites. We believe on the other hand that if we are to solve asynchronous interference locally, we cannot solve it across sites, due to the very asynchronous nature of networks which can be disconnected. Even though we want shared memory on a shared site, we want to prevent asynchronous modifications across sites.
Finally, we believe that agent does not have to be asynchronous on a common site, and that in fact we want to have a deterministic scheduling where agents have a fine-grained control of the scheduling. We are not going to discuss the security aspects of agent systems.
In order to fulfill these goals and requirements, we present ULM: a language designed from the start for mobile agents, with local deterministic scheduling and shared memory, and isolation from remote interference (inherently asynchronous) to a few well-known locations. The following chapter will describe the language, followed by the description of its implementation.

Chapter 3

Language

> *Si vous avez besoin de quelque chose, appelez-moi. Je vous dirai comment vous en passer.*
>
> – Coluche

> *I think perhaps the most important problem is that we are trying to understand the fundamental workings of the universe via a language devised for telling one another when the best fruit is.*
>
> – Terry Pratchett

The first things we have to do when presenting a new language is explain why it was created, what it consists of and how it works. In this chapter we are going to explain the origins of the ULM language, how and why it was created, what goals it is expected to accomplish. Then we will introduce gently (using running examples) the features that make ULM a new language. After such precautionary introduction to ULM, we will talk about the semantics more formally, first of the local thread scheduling, then dive into how mobile agents handle mobility.

3.1 Origins of ULM

ULM was introduced by Gérard Boudol [Bou04a] in 2004 as a programming model for mobile agents in a GALS (Globally Asynchronous, Locally Synchronous) network. This model provides threads, synchronisation mechanisms and mobile agents. One of the goals of this model was to isolate non-determinism to the places where it cannot realistically be avoided.
The local scheduling of threads in ULM is based on the work on Fair Threads by Frédéric Boussinot [INR] which provide deterministic scheduling of threads in a dynamic language. One of the goals of Fair Threads was to fix some of the problems programmers face when dealing with threads. In this section we are going to introduce ULM's programming model.

3.1.1 Local thread scheduling

One of those problems is the lack of semantics in thread scheduling. Very often thread models come with no scheduling semantics, and it is up to the programmer to protect

his code from scheduling interference. Fair Threads are scheduled deterministically. This means that we can both predict the outcome of the scheduling, and that all executions of the same code will be scheduled in the same way.
This is a very important property to have when writing code that uses threads: if the semantics are clear enough, the programmer will know exactly how to write threads that do what he wants.
The *Fair* in Fair Threads comes from the fact that the scheduler semantics guarantee that any thread which wants to be scheduled will get scheduled in due time in a fair manner, as long as the threads themselves play *fair* by *cooperating* as described below. To that end the scheduling time is divided in a logical abstraction of time called *instants*. An instant is a scheduling round where each thread is scheduled if it wants to and if it can.
The main communication and synchronisation means in Fair Threads is called a *signal*. A signal is an object which can have two states: *not emitted* and *emitted*. Each signal starts in the *not emitted* state. Threads can *emit* signals, thus changing the signal's state to *emitted* until the end of the instant, where each signal is reset to the *not emitted* state. Threads can also *wait* for a signal to become emitted. If the signal has already been emitted no waiting is done. Otherwise the thread is suspended until the first signal emission. Since the signals are kept emitted for the entire instant and each thread is given a fair execution slot during the instant, no thread can miss a signal emission.
The Fair Threads scheduling is called *synchronous* because of the time slicing with instants, in which threads are scheduled sequentially. This scheduling is also called *cooperative* because each thread has the responsibility to tell the scheduler when it is ok to schedule another thread before it is given back some scheduled time. Synchronous cooperative threads in the Fair Threads model are called *reactive* threads.
The fact that threads have to cooperate explicitly to allow the scheduling of other threads could possibly be seen both as the main advantage and disadvantage of this model. It is crucial for a thread to be able to decide when it wants to cooperate in order to keep a clear flow of code, and at the same time it is crucial for threads to cooperate at all. Otherwise the other threads' scheduling will be blocked. The correct locations of cooperation points and their frequency are some of the drawbacks we have to bear for the possibility to choose their location.

3.1.2 Migration

Having a model that allows a programmer to write threads that behave according to deterministic and clear semantics is great, but this is not really new. What is new in ULM is the integration between mobile agents and reactive threads.
Mobile agents are a special kind of thread that can move from one *site* to another. A site is usually a computer, but we can also have several sites on one computer. Being able to move a thread from one computer to another can be useful for all sorts of tasks: code distribution, load balancing, execution of remote procedures on dedicated computers, the list goes on and grows everyday as computers get cheaper and interconnected.
There are several kinds of mobility: being able to move code, being able to move data, and being able to move both. Being able to move code around is nothing new: it can be transferring a program from one computer to another, which was done using floppies even when computers where not networked together. What is more interesting is when new code is loaded from another computer into an already running program. But again this is not new: dynamic loaders have been around for a while.

Being able to move data from one place to another is again not a novelty. But together with being able to move code they provide the essential ingredients for thread mobility. Threads are composed of exactly these two things: code, state and data. Specifically, the state is the state of the thread and the data are whatever it needs to work on.
There are two kinds of thread mobility: *weak* and *strong* mobility. Weak mobility refers to moving a thread's code and data but not its state. A thread's state holds the information such as which function the thread is in, or what function it needs to return to. Generally a thread state can be seen as its execution stack, its continuation, the local variables' values, the processor or virtual machine registers: whatever data indicating what the thread is doing and has to do next.
Weak mobility generally involves saving the data that the thread needs to work on, saving its code, and sending that data to be loaded somewhere else. Because the thread state is lost, what the destination site will do is create a new thread, unpack the code, the data, and start the thread on a function that was specified before the migration.
Strong mobility on the other hand packs the thread state along the code and data, so when the thread is respawned on the destination site it picks up its execution with the same state it was before it left. The thread's current function stays the same, the local variables too, it will return to the same functions, etc.
Weak mobility can be seen as giving up on the thread's continuation, and invoking a function on a remote site in a new thread. Actually it also does handy packing of data and several other things, but as far as thread state is concerned that's it. This is why generally the thread's stack has to be empty to allow migration. Either that or the thread's continuation is lost.
With strong migration it is not the case: function A can call function B, which will cause migration before returning to function A.

3.1.3 Non-determinism isolation

We already explained the determinism of local thread scheduling, but what happens when we integrate migration into the scheduling? Migration involves network communications, or more generally communication between a site and another. Such communication is outside the scope of a site. In the case of network communication, it is inherently non-deterministic: links can fail, computers fail, and a single site cannot be responsible for or even know what happens remotely in places it cannot control.
Since ULM aims at limiting non-determinism, migration has been adapted to play nicely with the reactive thread model. All that happens within the instant is and remains deterministic. Agents should be sent and received by the scheduler between instants. This means that agents are synchronous and deterministic up until the time they are frozen and transported to another place where they become synchronous to the new site. For all other threads watching the scene, any agents present in a given instant stay synchronous for the entire instant.
As for what happens during the migration, nothing is guaranteed: two agents can leave one site at the same end of instant, and arrive on different instants to the same remote site.
In networked computation it is also useful to be able to communicate values from one site to another. But we have already shown how to communicate threads, which come with whatever data they need to do the job, so actually communicating values can be done this way. Any data communication supposed to happen within the instant between two

sites should not be possible, as we already explained why inter-site communication is non-deterministic. But to communicate a value to someone also means being able to give it to someone: having some other thread somewhere waiting for that data.
This is usually done with *network references*, the equivalent of having a handle, a pointer to a variable that is located on a different site. Allowing this sort of communication within the instant would break ULM's goal of isolating non-determinism. But allowing it between the instants would not make much more sense: imagine two threads on different sites, trying to read or write into a variable located on a third site. The local semantics cannot enforce any ordering, any determinism over what happens somewhere else.
But network references are necessary nonetheless, which is why ULM has them, but with a twist, so they are simply named *references*. References in ULM can point to data storage locations which can be allocated either on a site (their storage is fixed) or on an agent (their storage moves along with its agent). Whenever the reference points to a storage located on the same site, we call it a *local reference*, and reading and writing to it is permitted. This happens either when accessing a reference fixed to the local site, or located on an agent which we already explained will be there for the entire instant. Whenever the reference points to a storage located on a different site, we call it a *remote reference*, and any reading or writing to it will suspend the thread until the reference becomes local. The suspension happens whenever attempting to access a remote reference fixed to a remote site, or an agent which is not on the local site. The unblocking occurs when either the suspended thread is moved to the correct site, or when the remote reference comes to the local site.
ULM references may look strange at first, but it has the semantics of being able to point to a remote variable, while at the same time ensuring that any concurrent read or write to that variable is done by threads that are on the same site for the entire instant, and is therefore deterministic.

3.2 Features

Now that we are familiar with the origins of ULM's core model, we can take a look at how we embedded it in a language. In this section we will describe what ULM programs look like and consist in. We will do this by presenting every ULM feature in detail with examples, while discussing the rationale for each feature.

3.2.1 Scheme embedding

Although ULM's core model was presented for ML [SML], it was designed for any functional language. For the purpose of studying the model's introduction in a language we chose Scheme [KCe98] as the target functional language. Scheme is well known for the small size of both the core language and its interpreters, as well as being among the usual suspects for experimentation on languages.
The target Scheme we chose was Revised^5 Report on the Algorithmic Language Scheme R^{5}RS, although it was not our goal to support the full standard. Indeed we are building a prototype language in order to study the impact of agent mobility in a functional language as well as the benefits of the ULM core model. To that end we have chosen to implement the minimum of R^{5}RS as required by our study. This turned out to be most of R^{5}RS, and then some.
Whenever there was Scheme functionality needed we chose the R^{5}RS specification of that

```
(ulm:thread
 (let loop ()
  (display "Ping ")
  (ulm:pause)
  (loop)))
(ulm:thread
 (let loop ()
  (display "Pong\n")
  (ulm:pause)
  (loop)))
```

Figure 3.1: Threads

functionality if it existed, or took from one of the many Scheme Request For Implementation (SRFI)[1] in order to have our language as standards-compliant as possible. Sometimes yet, when a required functionality was defined neither in R^{5}RS nor in any SRFI, or if it was, but we were not satisfied with it, we chose to look at what Scheme's future new standard R^{6}RS [SCD$^+$] had to say about the matter. While it is not finished yet, and ULM did not target it (it wasn't even a draft when we started working on ULM), there is no doubt that it will become the standard Scheme reference in the coming years, so it is worth looking at.
Aside from the Scheme primitives and libraries needed to host the ULM core model, our work consisted in integrating the ULM primitives with Scheme. This implied figuring out which Scheme primitives were affected by ULM, as well as the other way around.
It is beyond the scope of this dissertation to describe the Scheme language where ULM does not affect it, therefore whenever standard Scheme language, primitives or functions are used they will not be described. Scheme primitives that are affected by ULM will be discussed further down this section.

3.2.2 Threads

We already discussed how ULM threads are cooperative and deterministic. We already discussed the notion of instants. We will now see our first ULM program and start illustrating what can be done with ULM. The program in Figure 3.1 creates two threads that together display the string "Ping Pong\n" forever.
In this program you can see the two first ULM primitives `ulm:thread` which creates a new thread in which to evaluate its body, and `ulm:pause`[2] which causes a thread to wait for the next instant. In ULM, when a new thread is created, it gets automatically scheduled later within the instant. Specifically it gets appended at the end of the list of threads that will be scheduled next within the instant. This means that creating a thread is a non-blocking operation.
In our example it means that each of the two threads is created and put in their creation order at the end of the scheduler's queue before any of them gets executed. In this specific example, assuming no other thread has been created than the thread executing our example, they will get executed when that thread terminates.

[1] A SRFI is a specification of functionality for Scheme grouped by theme: lists, exceptions...

[2] Although `ulm:pause` can be derived from other instructions, it has been promoted to a primitive because it is a pillar of scheduling optimisation (which we describe in 5.3.1) and ongoing work on static analysis.

In Scheme, expressions that are not in procedures are evaluated in what is called the toplevel. In ULM, no expression can be evaluated outside a thread, so the toplevel is actually evaluated in a primordial thread (one created by the system). This thread will terminate when the toplevel evaluation is finished.
We already discussed the fact that threads have to relinquish the control to the scheduler before any other thread can be scheduled: this is the cooperation point. The first cooperation point we see in this example is that when a thread terminates (the primordial toplevel-executing thread here), it gets removed from the scheduler and the next thread to be scheduled in the instant gets executed[3].
To get back to our example, let us look at the body of each thread (they only differ by the string they print). They enter an infinite loop where they display their string before waiting for the next instant by calling `ulm:pause`. This is the second cooperation point: when a thread waits for the next instant, it gets put in the list of threads to schedule at the next instant (specifically at the end of that list).
Let us list the events we have just described:

1. The primordial thread is created.
2. The primordial executes the toplevel.
3. The first thread is created.
4. The second thread is created.
5. The primordial thread terminates.
6. The first thread is scheduled.
7. The first thread prints “Ping ”.
8. The first thread waits for the next instant.
9. The second thread is scheduled.
10. The second thread prints “ Pong\n”.
11. The second thread waits for the next instant.

Now we reach a point where there are no more threads to schedule during the current instant (since both live threads are scheduled for the next instant). At this point the scheduler enters the *End Of Instant* (EOI) phase. Several things happen during that phase, and each will be described as we introduce new features. The first thing that happens at the EOI relevant to our example is that the list of threads to be scheduled for the next instant becomes the list of thread to schedule during the new instant that is started after the EOI. It is important to note that the list keeps its ordering, which means that threads are scheduled in the same order they waited for the next instant. More generally, threads in ULM are always rescheduled in the order they cooperate.
Let us now look at what happens at that next instant, right after the first EOI:

1. EOI.

[3]We will discuss later what happens when there are no more such threads.

2. The first thread is scheduled.
3. The first thread prints “Ping ”.
4. The first thread waits for the next instant.
5. The second thread is scheduled.
6. The second thread prints “ Pong\n”.
7. The second thread waits for the next instant.
8. EOI.
9. ...

This is exactly the same scheduling that happened at the previous instant. In fact since both threads are in infinite loops, the same scheduling will occur at each instant. This means that the result of this program will consist in printing “Ping Pong\n” once per instant for ever. The semantics of ULM's scheduling make it easy to write threads that are synchronised (one is always called before the other) in an intuitive way (creating them in the proper order is enough).

3.2.3 Signals

In the previous example we have shown how two threads are synchronised by both waiting for the next instant. This is enough for the second thread because at the next instant it knows the first one will have been scheduled before itself. But this type of synchronisation falls short of handling intra-instant synchronisation as well as being able to wait for several instants.
This is where signals come in handy: they are the main synchronisation means of ULM. As described earlier, signals have two states: non-emitted and emitted. They start as non-emitted, and are reset to that state at each EOI. When emitted, they keep that state until the EOI. With signals, threads can wait for a signal to become emitted (if it isn't already), whether that emission will be later in the same instant or in several instants.
Signals are created with `ulm:signal`, emitted with `ulm:emit` and waited for with `ulm:await`.
In order to introduce signals, let us take the previous example of two threads printing “Ping Pong\n” and introduce a bug in Figure 3.2.
As far as both threads are concerned nothing changes except they now call our function `disp` instead of `display`. Our function will randomly call (or not) `ulm:pause` to wait for the next instant, before printing. So in fact this function may or may not skip instants. The result is that sometimes the first thread skips an instant, resulting in the printing of “Pong\n”, sometimes the second thread skips an instant (printing only “Ping ”), sometimes they print “Ping Pong\n” correctly, and other times they both skip an instant before printing correctly.
Thus our contract is broken. While it may seem unfair to introduce randomness consciously in our program, we do that to illustrate that when calling functions it may or may not be obvious what scheduling changes they introduce. Suppose that `disp` was defined in a library written by a third-party, and that library implements non-blocking Input/Output by cooperating instead of blocking. This is not a far-fetched idea, but it means that for some external reasons we might skip instants, or not.

```
(define (disp arg)
 (if (= (random 2) 0)
  (ulm:pause))
 (display arg))

(ulm:thread
 (let loop ()
  (disp "Ping ")
  (ulm:pause)
  (loop)))
(ulm:thread
 (let loop ()
  (disp "Pong\n")
  (ulm:pause)
  (loop)))
```

Figure 3.2: Randomly skipping instants

In order to be sure that the first thread has been scheduled before the second thread in such a case, we need signals. Let us rewrite the previous example using a signal emitted by the first thread and awaited by the second as shown in Figure 3.3.
What we added here is that we create a signal named `relay`, which is emitted by the first thread after printing, and awaited by the second before printing. Both threads still need to call `ulm:pause` to switch instant in order to reset the signals. Otherwise the first thread would never cooperate and try to emit the signal several times during the same instant (only the first emission has any effect), and the second thread would wait several times in the same instant for a signal that would stay emitted, thus not cooperating.
So now we are sure that the first thread has displayed its string before the second does, but there still is a bug in there. It happens if the second thread takes several instants to display, it means the first thread can redisplay its "Ping " before the "Pong\n". So we have to use another signal so that the second thread can notify the first thread when it can resume, as shown in Figure 3.4.
This time we use two signals, `ping` for the first thread to notify the second it can print, and `pong` for the second thread to notify the first when it can print. This example is a good example of why introduction of non-determinism in a thread can complicate the scheduling, and how signals are an appropriate means to synchronise thread even in such a case.

3.2.4 Suspension

Now that we understand ULM's scheduling and signals, we can look at the suspension control block. In ULM it is possible to condition the evaluation of an expression on the presence of a signal. It means that the expression will be suspended by the scheduler at each instant until the given signal is present, or until the expression returns. This is done with the `ulm:when` primitive, whose first argument is the signal to suspend on, and the other arguments are expressions to evaluate only when the signal is emitted. The return value of `ulm:when` is that of its last evaluated expression.
For example, we will try to have two threads printing a message on a different clock: one

```
; (disp arg) stays the same

(define relay (ulm:signal))

(ulm:thread
 (let loop ()
  (disp "Ping ")
  (ulm:emit relay)
  (ulm:pause)
  (loop)))
(ulm:thread
 (let loop ()
  (ulm:await relay)
  (disp "Pong\n")
  (ulm:pause)
  (loop)))
```

Figure 3.3: Using a signal

```
; (disp arg) stays the same

(define ping (ulm:signal))
(define pong (ulm:signal))

(ulm:thread
 (let loop ()
  (disp "Ping ")
  (ulm:emit ping)
  (ulm:await pong)
  (ulm:pause)
  (loop)))
(ulm:thread
 (let loop ()
  (ulm:await ping)
  (disp "Pong\n")
  (ulm:emit pong)
  (ulm:pause)
  (loop)))
```

Figure 3.4: Using two signals

```
; this prints msg at every instant
(define (loop-print msg)
 (let loop ()
  (display msg)
  (ulm:pause)
  (loop)))

(define two (ulm:signal))
(define three (ulm:signal))

(ulm:thread
 (ulm:when two
  (loop-print "2 instants")))

(ulm:thread
 (ulm:when three
  (loop-print "3 instants")))

; emit the signals
(let loop ((i 1))
 ; every two instants
 (if (= 0 (modulo i 2))
  (ulm:emit two))
 ; every three instants
 (if (= 0 (modulo i 3))
  (ulm:emit three))
 (ulm:pause)
 (loop (+ i 1)))
```

Figure 3.5: Using suspension

will print at every three instants while the other will print every other instant. The code is shown in Figure 3.5.
In this example we have delegated the infinite loop printing to the function `loop-print`. We then have two signals: `two` is emitted by the primordial thread every two instants, while `three` is emitted every three instants. We also have two threads, both entering a suspension context[4] on one signal, in which they call `loop-print`.
Let us trace what happens in this example:

1. The primordial thread creates two threads, emits no signal and cooperates.
2. The first thread starts and gets suspended on `two`.
3. The second thread starts and gets suspended on `three`.
4. End of instant 1.

[4]We use the word *context* to denote the list of preemption and suspension blocks in a thread's dynamic extent.

5. The primordial thread emits `two` and cooperates.
6. The first thread is resumed, enters `print-loop`, prints and cooperates.
7. End of instant 2.
8. The primordial thread emits `three` and cooperates.
9. The second thread is resumed, enters `print-loop`, prints and cooperates.
10. End of instant 3.
11. The primordial thread emits `two` and cooperates.
12. The first thread is resumed, prints and cooperates.
13. End of instant 4.
14. The primordial thread emits no signal and cooperates.
15. End of instant 5.
16. The primordial thread emits `two`, `three` and cooperates.
17. The first thread is resumed, prints and cooperates.
18. The second thread is resumed, prints and cooperates.
19. ...

One of the powerful features of the suspension primitive is that the code evaluated in its body does not have to be modified to be scheduled according to the suspension signal: at every new instant the body will wait again for the signal to be emitted. This allows us to have the same function called from both threads behave differently depending on the suspension context.
Now that we explained how the suspension primitive works, we can confess that `ulm:await` can be defined in terms of `ulm:when`[5]:

```
(define (ulm:await sig)
 (ulm:when sig #t))
```

As you can see waiting for a signal consists in entering a suspension block on that signal, and as soon as it is emitted, return from the block with any value.

3.2.5 Weak Preemption

We now know how to suspend the evaluation of an expression at every instant, for example to constrain a certain evaluation to the availability of a given resource. But what happens when such a resource is determined to be absent for ever on? Sometimes suspending is not enough and we would like to give up execution of a certain expression in the middle of its evaluation. Moreover it would be nice if that expression did not have to be rewritten to check every so often whether it should stop or not. On the other hand, we do not want the

[5]Even though for performance reasons we promoted it to a primitive in our implementations.

```
(define sig (ulm:signal))

; enter weak preemption on sig
(ulm:watch sig
 (print "one")
 ; trigger the preemption
 (ulm:emit sig)
 ; wait for the end of instant to
 ; be preempted
 (ulm:pause)
 (print "two"))

(print "three")
```

Figure 3.6: Weak preemption

```
(define sig (ulm:signal))

(ulm:thread
 (ulm:watch sig
  (print "one")
  (ulm:emit sig)
  (print "two")))
```

Figure 3.7: Subjective weak preemption

traditional preemption wherein the system preempts unknowing threads at various random (and often unexpected) places.
The second (and last) control block in ULM allows the programmer to solve all that. The `ulm:watch` primitive takes a signal as first argument, and evaluates its body until it either terminates normally, or until the end of the instant when the signal is emitted. If the signal is emitted and the expression does not exit the preemption block before the end of the instant, then the expression is given up and `ulm:watch` returns at the next instant. The preemption primitive returns no value. Figure 3.6 shows an example of weak preemption:
This example will cause “one” and “three” to be printed, because when the thread reaches the end of instant with `ulm:pause` it will be preempted by the weak preemption block whose signal has been emitted.
The fact that the preemption takes place at the end of the instant of the signal emission is called *weak* preemption. It is there because signals can be emitted by every thread, thus preemption of a thread can be triggered either by the thread itself (*subjective* preemption), or any other thread (*objective* preemption). The code in Figure 3.7 illustrates this:
In this example we create a signal and a thread. That thread enters a preemption block on the signal where it prints, emits that signal and prints again. Let us suppose that preemption should occur at the moment of the signal emission rather than at the end of instant. This would be *intra-instant* preemption, or *strong* preemption. It would mean emitting the signal would preempt the thread and cancel the second printing. But because

```
(define sig (ulm:signal))

(ulm:emit sig)

(ulm:thread
 (ulm:watch sig
  (print "one")
  (ulm:emit sig)
  (print "two")))
```

Figure 3.8: Objective weak preemption

any thread can emit the signal, let us see what happens if another thread emits the signal as shown in Figure 3.8.

Now the primordial thread has emitted the signal before the new thread enters the preemption block. If we want to be consistent with the form of strong preemption we are trying in this example, the body of the preemption block should not even be entered, since the signal is already emitted. This makes for a different scheduling within the same instant depending on when the signal was emitted.

This breaks the contract of instants and signals which specify that signals that get emitted should be considered emitted for the entire instant, and makes it hard for the programmer to understand the scheduling. When the end of instant is the only moment when preemption takes place this problem is solved and the moment where a signal is emitted within an instant does not modify the scheduling of preemption blocks.

It is important to understand how ULM's weak preemption is different from both exceptions (the traditional `try/catch`) and Pthread's [IEE95] `cancel`. The usual exception mechanisms does indeed allow a thread to preempt a certain evaluation, and would be equivalent to the strong preemption we just described, but only the thread itself could preempt itself: the preemption could not be triggered by others as can be done in ULM.

POSIX threads on the other hand do allow any thread to preempt one another (even though the exact place where the preemption occurs is non-deterministic): POSIX calls it *canceling*. The big difference with ULM is that canceling a thread in POSIX means giving up the entire thread, there is no nesting of preemption blocks, it is only possible to kill a thread, not merely making it give up on one expression.

Let us look now at a useful application of the preemption block. Suppose we want to determine whether a certain signal has been emitted within the instant. Merely waiting for it can only tell us that it was emitted if and when it is emitted. The waiting could simply never return if the signal is not emitted. On the other hand we cannot rule out that a signal has not been emitted until the next instant. Suppose we could tell that a signal has not been emitted yet during the current instant: what stops a thread to emit it after we have determined it hadn't been emitted? An authoritative answer on the absence of signal emission can only be given at the next instant, when we can go back and look at the previous instant to say: that signal was never emitted. On the other hand emission of a signal can be determined instantly, so we would like to write a function that either returns `#t` within the instant if the given signal is emitted, or `#f` at the next instant if it isn't.

In order to write such a function, we first need another function which executes an expression

```
(define (ulm:now thunk)
 (let ((kill (ulm:signal))) ; a local signal
  (ulm:emit kill)
  ; entering a preemption block on an emitted signal guarantees
  ; its body will only be executed at most during this instant
  (ulm:watch kill
   (thunk))))
```

Figure 3.9: One-instant execution

```
(define (ulm:present sig)
 (let ((ret #f)) ; our return value
  (ulm:now
   (lambda ()
    ; await the signal's emission
    (ulm:await sig)
    ; the signal was emitted during this instant
    (set! ret #t)))
  ; ret is #t if that signal was emitted, otherwise it was left at #f
  ret))
```

Figure 3.10: Signal presence check

for at most one instant. This is done with preemption by executing an expression in a preemption block on a signal which we emit ourselves prior to entering the preemption block. This is called a *surefire* preemption, because we know the preemption will be triggered unless the expression terminates before the end of instant. This function is shown in Figure 3.9. We can now write our `ulm:present` function to determine a signal's presence during the current instant, or its absence at the next. This function uses `ulm:now` to wait for the signal we are testing during at most the current instant. If the signal is emitted during the instant, we return `#t`. If it was not emitted and our wait for the signal was preempted by `ulm:now` we return `#f` at the next instant. This function is shown in Figure 3.10.

3.2.6 The integration of weak preemption in Scheme

As promised, we discuss Scheme primitive when they interact directly with ULM's primitives, whether by being affected or by affecting them. Weak preemption is one of ULM's features which interacts directly with several Scheme primitives.

Subjective strong preemption with call/cc

Subjective preemption can be implemented in Scheme by using the traditional continuation capture primitive `call-with-current-continuation` (`call/cc` for short). This primitive allows the programmer to save the continuation at a given point and later invoke it. This is a complex primitive whose interaction with the rest of Scheme is also complex, and this is the only place where we will discuss it. When invoked within its dynamic extent, `call/-cc`'s reified continuation has the effect of *unwinding* back to `call/cc`'s call location. This

is a form of strong subjective preemption. It is strong because the unwinding happens as soon as the continuation is called (not at the end of instant), and it is subjective because the thread which invokes the continuation is the one which will unwind. Because it is not possible for a thread to force another thread to invoke a continuation, it is not possible to do objective preemption with `call/cc` like we can do with `ulm:watch`. On the other hand strong subjective preemption can be useful in some cases.
But `call/cc` also provides more than just strong subjective preemption: it allows continuations to be invoked several times, or to escape the dynamic extent of `call/cc`. When threads are involved things get bizarre though, because it may not be clear what happens when a thread invokes the continuation of another thread. There has been studies on this subject however [MGK03] and an intuitive implementation of a continuation in ULM could be to save the preemption and suspension context on top of the traditional scheme continuation and invoking another thread's continuation would substitute the current continuation (thus giving up on any pending preemption) with a copy of the other thread's continuation, possibly being blocked by suspension contexts.
But as a programmer, multi-shot continuations (continuations that can be called several times) as well as continuations that replace the stack (as opposed to exception which only unwind it) require a certain attention throughout the program. Each function has to be aware that its execution can be saved and restored later, possibly by a new thread, and certain dispositions have to be taken to keep the code working in such cases (using `dynamic-wind`), when it is even possible at all.
We have not yet talked in detail about agents, but for now let us simply state they are a special kind of thread. Entering a function as an agent and leaving it as a normal thread does not seem like something simple to sort out, yet it would be possible using `call/cc`. A solution to this particular problem could be to limit calls to continuations to the thread (or agent) that created it, but then we are putting limits the use of `call/cc`.
We believe `call/cc` is too powerful for most programmers to yield safely. We also believe that we can offer other ways to implement most of what it usually allows. In ULM threads do not have to be implemented using continuations: they are part of the language. Coroutines can be implemented using generators instead of continuations. Backtracking can be done without `call/cc` using CPS (Continuation Passing Style). For all these reasons we decided not to integrate it in ULM.

Strong subjective preemption without call/cc

We have seen earlier examples of why objective strong preemption leads to problems, but since `call/cc` only allows strong *subjective* preemption, we have no problem anymore with it being *strong* rather than weak. This is one feature we really like and cannot do with ULM's core model. Only a small part of `call/cc`'s features is needed to implement this though: being able to unwind the stack to a given point just like exceptions. Bigloo [SW95] has such a variant of `call/cc` called `bind-exit`. It takes one function as argument and calls it with its current one-shot unwind-only continuation as argument.
This means that the continuation is only valid once, and within the dynamic extent of the call to `bind-exit`. We have integrated this function in ULM, with the logical addition that calling the continuation made available restores the preemption and suspension context that was in place around the call to `bind-exit`, since it should mean the same thing as letting it return normally.
While we integrated this function in ULM, we noticed that a mechanism was missing

```
(define (with-input-from-file string thunk)
 (let ((port (open-input-file string)))
  (if (input-port? port)
   (let ((old-input-port (current-input-port)))
    (unwind-protect
     (begin
      ; we modify the current input port by side-effect
      (current-input-port-set! port)
      (thunk))
     (begin
      ; we restore the current input port before exiting
      (current-input-port-set! old-input-port)
      (close-input-port port))))
   (error "with-input-from-file" "can't open file" string))))
```

Figure 3.11: A safe with-input-from-file

from the core model: a feature called *finally* in many exception mechanisms, and called `dynamic-wind` in Scheme. It refers to the ability to execute some expressions as we unwind the stack as a result of preemption. `dynamic-wind` also allows to execute some expressions as we restore a continuation that had been exited with `call/cc`, but this is a feature we do not need. It is different from *catch* in exception mechanisms: it means that in the event of preemption, we want to execute some code before the preemption goes all the way up the stack.
`dynamic-wind` is not meant to block the preemption like *catch* or even as a final target for preemption like `ulm:watch`, but as an obligatory passing point before leaving a function. It is necessary in many places where a certain side-effect has to be cleaned up before exiting the function, whether or not the function is exited normally or by being preempted.
In Bigloo the equivalent of `dynamic-wind` is called `unwind-protect` (because it only protects from unwinding) so we have kept that name in ULM. It is the equivalent to *finally* and is called a protection block. An example of how Scheme's `with-input-from-file` could be implemented safely with regards to preemption is shown in Figure 3.11.
The semantics of strong subjective preemption in ULM follows logically: when unwinding the stack because of strong subjective preemption (with `bind-exit`) the preemption and suspension context is restored accordingly as we go up and execute all the protection blocks until we exit the target preemption block. And when unwinding the stack as a result of weak preemption the suspension and preemption contexts are also restored accordingly as we go up, but all the protection blocks are called at the instant following the weak preemption since preemption is decided at the EOI.

3.2.7 The interaction between when, watch and finally

Preemption, suspension and protection are quite straightforward by themselves, but when they are combined they interact in ways which, although logical, are not necessarily intuitive at first.
The first thing to understand about suspension is that when an expression is suspended it is not only not scheduled: it is simply not taken into account by the scheduler. The code

```
(define s1 (ulm:signal))
(define s2 (ulm:signal))

(ulm:thread
 (ulm:when s1
  (ulm:await s2)))

; first instant
(ulm:emit s1)
(ulm:pause)
; second instant
(ulm:emit s2)
(ulm:pause)
; third instant
(ulm:emit s1)
```

Figure 3.12: Suspended waiting

```
(ulm:thread
 (ulm:when s1
  (ulm:when s2
   #t)))
```

Figure 3.13: Suspended waiting (variant)

in Figure 3.12 illustrates that.
In this example we have two signals `s1` and `s2`. We create a thread which will be waiting on the latter in a suspension block on the former. During the first instant we emit `s1` in order to allow the thread to enter the suspension block and start waiting for `s2`, which is not emitted during the first instant. At the second instant we do emit that signal, but because `s1` is not emitted at this instant, our thread is suspended and the emission of `s2` has no effect at all on the thread. At the third instant we unfreeze our thread by emitting `s1` but it will still be waiting for `s2`.
When we consider the fact that `ulm:await` is derived from `ulm:when`, the previous thread can be re-written as in Figure 3.13. It becomes clear that `#t` can only be returned during instants where both signals are emitted.
Preemption blocks are also affected by suspension blocks: preemption can only be triggered when the preemption block is not suspended. The example in Figure 3.14 illustrates a non-suspended preemption.
On the other hand, because any expression within the dynamic extent of a suspended `ulm:when` is ignored by the scheduler, if the preemption block is suspended at the instant when preemption would occur, it simply does not occur, as shown in Figure 3.15.
Because the emission of `s2` happens during an instant where the preemption block is suspended on `s1`, the wait on `s3` is never preempted and the thread is blocked forever.
Now if we mix protection into this example we get the last tricky example. In Figure 3.16 we emit `s1`, get into a preemption block on it, then into a suspension block on it before

```
(ulm:emit s1)
; this is a sure preemption: we wait for
; at most 1 instant
(ulm:watch s1
 (ulm:when s2
  #t))
```

Figure 3.14: Non-suspended preemption

```
(ulm:thread
 (ulm:when s1
  (ulm:watch s2
   (ulm:await s3))))

; fist instant: allow the thread to enter watch and await
(ulm:emit s1)
(ulm:pause)
; second instant: preempt the await ?
(ulm:emit s2)
(ulm:pause)
; third instant: nothing happened, the thread is still blocked
```

Figure 3.15: Suspended preemption

we enter a protection block where we wait for a non-emitted signal `s2`. At the end of the first instant, the preemption is triggered and at the second instant the thread should get out of the preemption block. For this it needs to execute all the protection blocks up until the preemption target. In our case we have installed a protector which should print when protected. But the trick is that this protector is under a suspension block on `s1` which is not present at the second instant. Therefore the protector will be suspended until `s1` is emitted again. This is logical since the protection block is inside the suspension block, even though it is not obvious since it prevents us from getting out of the preemption.

3.2.8 Exceptions

We are now familiar with preemption (weak and strong) and protection in ULM. This means we can talk about exceptions and errors, and for that we need to talk about *standard* Scheme exceptions and errors (or lack thereof).

R^5RS has a very loose meaning of what an *error* is, even though it mentioned throughout the standard that errors should occur. SRFI-23 [23] (Error reporting mechanism) introduces the `error` procedure which triggers an error, but does not specify what *triggering* means except in the terms that it should be the same error triggering than produced by Scheme internally (R^5RS errors) and it should be displayed somehow to the user.

Suggestion is made in SRFI-23 that triggering an error could be implemented as raising an exception of the *error* kind. Exceptions are yet another mechanism not described by R^5RS, although they are described in SRFIs 34 [34] (Exception Handling for Programs) and 35 [35]

```
; emit s1 so we wait for s2 at most one instant
(ulm:emit s1)
; surefire preemption
(ulm:watch s1
 ; we get in ok since s1 is emitted
 (ulm:when s1
  (unwind-protect
   ; this is what we protect
   (ulm:await s2)
   ; and this is how we protect it
   (print "preempted\n"))))
```

Figure 3.16: Suspended protector

(Conditions). The latter defines mechanisms for creating and composing exceptions (called *conditions*), while the former describes the framework for installing exception handlers and raising exceptions.

Traditional exceptions and handlers (by traditional we mean those used in today's most widespread languages) such as Java [JSGB00], C++ [Str00], C# [ECM01], Ruby [Rub] or Python [Pyt] provide exceptions at the language level, and each invoke exception handlers as the stack is unwound until the exception is caught. Exactly what we do with protection blocks and preemption. But in SRFI-34 it is specified that handlers should be called in the dynamic extent of the call that raised the exception. This means no stack unwinding. This is actually more powerful than traditional exception handlers since it is possible to treat the exception at the exact place where it occurred. And since Scheme provides stack unwinding via other primitives (`call/cc` in Scheme and preemption in ULM) building on SRFI-34's definition of exception handling we can easily obtain traditional exceptions.

But SRFI-34 is vague concerning the continuation to `raise` (the function that raises exceptions) after it has invoked the exception handlers. Since it is possible that handlers do not unwind, it is possible that calls to `raise` return and that does not seem like the most wanted error treatment. For example, if `error` is defined as raising an exception of *error* type, calling `error` can have no effect if the current handler decides not to do anything about it, leading in continuing the evaluation of code that yielded the error, possibly because its continuation was deemed impossible.

To that end, R^{6}RS incorporates SRFIs 34 and 35 in the standard library, but defines that should the current exception handler return to `raise`, a *non-continuable* exception is raised again to the next exception handler, until one handler does not return, or the default handler (specified as not returning from non-continuable exceptions) is reached. This makes more sense in most cases where code raising an exception expect to unwind. In order to keep the possibility of raising continuable exceptions, a new `raise-continuable` procedure has been introduced, which allows handlers to return.

This is the dual system of exception handling we chose for ULM: handlers and raise procedures like R^{6}RS, exceptions as defined by SRFI-35 and errors as defined by SRFI-23 by raising non-continuable exceptions. On top of the procedure that installs a new exception handler (`with-exception-handler`) we provide an equivalent procedure `with-exception-catcher` which sets the handler's continuation to that of the call to `with-exception-`

`catcher` in order to facilitate creation of handlers that behave like traditional exception catchers.
In terms of integration with ULM, because exception handlers do not unwind and exception catchers unwind using ULM's strong preemption, they do not require any adaptation themselves, or from ULM.

3.2.9 Migration

Now that we know everything[6] about local thread scheduling, here comes the interesting part[7]: migration.

Agents

We already discussed that we have a special kind of thread called an agent which can migrate between sites at the EOI. Let us go into the details of what an agent is, and how it differs from the threads we have already seen. The first difference we will see is that agents have names. These names are provided by the system upon their creation, and they are used to identify the agents for objective migration. In other words, an agent is not reified, but can be uniquely identified by its name, which is convenient because name values allow easy referencing to agents across sites without worrying about the location of the referenced agent.
As we can see from the following example, agents are created with the `ulm:agent` special form, which returns the new agent name, and takes an extra argument to `ulm:thread`, which is the name of the variable which will receive the agent name in the agent's body. Agents are scheduled just like normal threads.

```
(define name
 (ulm:agent (my-name)
  (print "Agent's name is: " my-name)))
```

Now that we know how to create agents, and how to get their names, let us use those names to migrate the agents. We already discussed the fact that agents migrate from one site to another between instants, that is: they leave one site at the end of an instant, and arrive at their destination site at the beginning of one of the new site's instant. Migration is triggered by the `ulm:migrate-to` primitive which takes as first argument the name of the target site, and as optional second argument the name of the agent to migrate (objective migration). If such a name is not given it defaults to the current agent's name (subjective migration). An error is thrown if attempting subjective migration from a thread which is not an agent[8].
The name of the target site consists in a string denoting the target site's address either as a TCP host (name of IP address, with possibly a port number) or as a bluetooth address. Let us suppose we have two ULM platforms, running on hosts A and B. Figure 3.17 shows how we would send an agent to print from host A to B.
Since migration occurs at the end of instant, calling `ulm:migrate-to` only has the effect of marking the current agent as a scheduled migration. It is a non-blocking operation, and

[6] Well, almost.
[7] If you thought the rest was already interesting, you're in for a treat.
[8] Actually when we see migration groups in 3.2.9 we will revise this statement.

```
(ulm:agent (my-name)
 (print "Agent is on host A")
 (ulm:migrate-to "B")
 (ulm:pause)
 (print "Agent is on host B"))
```

Figure 3.17: Migration

that is why the agent has to wait for the next instant using `ulm:pause` in order to physically move the agent to B, where it will return from `ulm:pause` to print.

Safe migration

Migration with `ulm:migrate-to` is all ULM's core model defined for migrating a thread, but this assumes that the transport was successful and that the other site was listening and accepted our agent. This is clearly not enough since transport can never be safe: sockets break, computers shut down, ULM platforms may not be running on the target site... This type of migration is *unsafe*, and it can be a good thing sometimes. But some other times we may want a *safer* type of migration.
There are two complementary things we need when we want to migrate safely: notification of success or failure of the migration, and be able to instruct the ULM platform what to do in case of failure. Because knowing that a migration failed is not enough: we also need to choose the best course of action for recovery of the agent. In some cases losing the agent during migration is not a problem, but in some other cases it might be necessary to restore (we call this *respawning*) the agent on the departure site because we cannot afford to lose it.
In order to handle these problems we added a new primitive `ulm:safe-migrate-to` which takes two additional arguments to `ulm:migrate-to`: a timeout (in seconds[9]) and a boolean to indicate whether or not to respawn the agent in case of failure. This primitive returns a signal which will be used to indicate the success or failure of the migration. This signal is special with regards to other signals for several reasons. Firstly it is emitted by the scheduler itself at the beginning of an instant when the scheduler has determined the success or failure of the migration in question. We call this type of signal *spontaneous* because it does not need to be emitted by any thread. Secondly if this signal is waited for using `ulm:await`, this primitive will return in case of migration success, and throw an exception in case of migration failure.
Such an exception is not thrown if the signal is waited for using `ulm:when` as this primitive is never *leaf*. We call a leaf primitive one on which a thread can be blocked at the end of instant at the top of the call stack. So far we have seen only two such primitives: `ulm:pause` and `ulm:await`, which both permit to change instants before they return. Although `ulm:when` also makes it possible to change instants, we do not call it leaf because it may not be on top of the call stack, as shown in Figure 3.18.
This illustrates that it is possible to be suspended in `ulm:when` in a leaf primitive which has nothing to do with the suspension signal (we could even be waiting for yet another

[9]Traditionally, FairThreads like to represent time in instants instead of *real* time. We find network timeouts to be easier to estimate in seconds than in instants, especially for interactive programes.

```
(define sig (ulm:signal))
; emit the signal for the first instant
(ulm:when sig
 ; now skip an instant
 (ulm:pause)
 ; we will never return from ulm:pause
)
```

Figure 3.18: `ulm:when` is not leaf

signal). If the previous example's signal was a signal migration it would not be clear where the exception should be triggered, since at the second instant we would be in `ulm:pause` waiting for the signal emission. If the exception was to be thrown by `ulm:pause` because of its suspension context, it might not be properly handled because it is not visually obvious that `ulm:pause` could throw exceptions (it would be worse even if the leaf primitive in this case was waiting for a second signal). On the other hand, throwing the exception at the level of `ulm:when` would not make sense since it would ignore everything in the suspension block. Whereas if a thread is waiting for that signal in a leaf primitive like (`ulm:await`) it is obvious that should the exception be thrown, this place is the right place to have a handler for it.

Figure 3.19 shows an example of how one might write a remote procedure call (RPC) which will try to execute the procedure on *fall-back* sites should the first one become unavailable.

In this example we iterate over a list of sites to try to send our thunk to. To that end we create an agent which will be sent to the first site to try. We use objective safe migration to try send the agent to the first site, and we order the ULM platform to ditch the agent in case of failure. We then install our exception handler and wait for the migration signal to be emitted. In case of successful migration, the waiting completes and we return `#t`. Otherwise an exception is thrown and caught by our handler which will try the next site until they have all been tried.

In the case of subjective migration, it is interesting to note that it does not make much sense to ask for safe migration without respawning, otherwise waiting on the migration signal will succeed only in case of success, or never return in case of migration failure, which really is the same as unsafe subjective migration. Figure 3.20 shows how one would write a function which does safe subjective migration and returns `#t` in case of success, or `#f` for failure (and obviously respawn).

Instructing an agent to migrate more than once in a given instant. Indeed, we take into account the first migration order (that is, the destination and possibly other safe migration parameters) and raise an error on subsequent calls to make the same agent migrate. If we were to do otherwise, like for example, honour the last migration order within an instant, we would not be able to instruct previous migration orders of the cancellation of their migration orders. As such, there cannot be conflicting orders for migration for the same agent (different sites, timeouts or respawn orders).

```
(define (rpc thunk sites)
 ; try every site
 (let loop ((sites sites))
  ; if we run out of sites to try, we failed
  (if (null? sites)
   #f
   ; make an agent in order to send the thunk
   (let* ((agent (ulm:agent (name)
                  ; wait for arrival since we are migrated
                  (ulm:pause)
                  ; now call the thunk on the new site
                  (thunk)))
          ; now get a signal for safe objective migration of our agent
          (signal (ulm:safe-migrate-to (car sites) ; the site
                                       10 ; timeout in ten seconds
                                       #f ; no respawn
                                       agent))) ; the agent to migrate
    ; now wait for success or failure
    (with-exception-catcher
     ; the handler
     (lambda (exc)
      ; migration failed: try a new site
      (loop (cdr sites)))
     ; the body
     (lambda ()
      ; if this does not throw it's a success
      (ulm:await signal)
      ; success
      #t))))))
```

Figure 3.19: Remote procedure calls in ULM

```
(define (ulm:migrate/wait site)
 ; trigger migration, and get its signal
 (let ((signal (ulm:safe-migrate-to site
                                    0 ; no timeout
                                    #t))) ; we want respawn
  ; prepare to catch migration failure
  (with-exception-catcher
   (lambda (exc)
    ; failure
    #f)
   (lambda ()
    (ulm:await signal)
    ; success
    #t))))
```

Figure 3.20: Safe migration

Groups

Another difference between threads and agents is that in fact threads are *attached* to the agent which created them (or the agent which created the thread which created them[10]). In fact, each thread has a parent: if it is created by an agent, that agent becomes its parent. If it is created by a thread, its parent is inherited. Threads that are created by the toplevel have no parent. Agents have no parent.
Threads with no parent are called *unmovable* threads, whereas thread with an agent parent are called *linked* threads. In fact they are linked to their agent parent, with which they form a *group*. If the agent migrates to another site, the whole group migrates with it. This is a useful feature since agents often need to have a group of threads with which they interact. Together they form a coherent group of inter-dependent threads, which is why migrating groups makes sense.
Figure 3.21 shows a rerun of our ping-pong example with one agent printing "Ping" and a thread in its group printing "Pong". Clearly they cannot function without each other, so they are migrated together.
Because they migrate together at the end of instant, they would print "Ping Pong" on each site they visit.

3.2.10 References

As previously discussed, it is possible in ULM to allocate a reference which will stay bound even when either the reference handle, or the referenced data become remote. References actually consist in a handle pointing to a memory slot in a memory store. Memory stores come in two flavours: site stores (which are always fixed to their site) and agent stores (which always follow the agent they belong to). This is the final difference between agents and threads: agents carry their store, while threads use the store of their group (whether from the agent parent or the site store for threads with no agent parent).

[10] ...or the agent which created the thread which created the thread which...

```
(ulm:agent (name)
 ; create our auxiliary thread
 (ulm:thread
  (let loop
   (print " Pong\n")
   (loop)))
 ; and print ping
 (let loop
  (print "Ping")
  ; migrate at each instant
  (ulm:migrate-to (random-site))
  (loop)))
```

Figure 3.21: Ping-pong with agent groups

References are created using the `ulm:ref` primitive, which takes the initial reference data as argument, and returns a new reference to a slot in a memory store as determined such: if the caller is an agent or a thread grouped to an agent, that agent's store is used, otherwise the site's store is used.
Accessing a reference's data (*dereferencing* it) is done using the `ulm:unref` primitive, which takes a reference as argument and returns the referenced data. Setting the reference's data is done using the `ulm:ref-set!` primitive, which takes a reference and the new data as first and second arguments.
Both accessing and setting a reference's data is a potentially blocking operation when the reference's memory store is located on a remote site. Figure 3.22 shows an example of how references are used in a non-blocking way to return a value from a remote procedure call. In this example, we create an agent which is sent by the caller thread to a remote site in order to execute a thunk there and bring back its return value. In the previous RPC example, we already saw how to send the agent to execute the thunk using objective migration. What this new version shows is the combination of a reference and a signal to communicate the thunk's return value back to the caller thread. The reference is created using the caller thread's store, and affected by the agent once it has migrated back from the remote site. The signal is also used by the agent in order to wake the caller thread which is waiting for it, thus notifying completion of the RPC. Once that signal is sent by the returned agent, and received by the calling thread, the thunk's return value can be read from the reference and returned to the caller.

3.2.11 Mixins

When it comes to user data types, R^{5}RS Scheme does not offer offer anything. In fact practically every Scheme implementation comes with its own set of user data type flavours, from plain structures to object systems. SRFI 9 (Defining Record Types) [9] proposes a standard way for defining user structures named *records* which are first-class and provide accessor, modifier, predicate and constructor functions as well as instrospection.
This type of record is further expanded in SRFI 57 (Records) [57], which adds syntax such as labels to access the record fields or function polymorphism for such records. R^{6}RS defines yet another type of record, with single inheritance among other features. Bigloo supports

```
; execute the given thunk on the given dest site
(define (ulm:rpc dest thunk)
 (let* ((sig (ulm:signal)) ; signal used to notify the agent's return
        (ret (ulm:ref #f)) ; the reference used to store the
                           ; thunk's return value
        (here (ulm:site-address)) ; the current site address, so we can
                                  ; come back
        ; we create an agent to go to dest
        (name (ulm:agent (name)
                ; at the next instant we will have been migrated
                (ulm:pause)
                ; store the thunk's result in a local variable because
                ; the reference would block here
                (let ((store (thunk)))
                 ; migrate back where we started
                 (ulm:migrate-to here)
                 (ulm:pause)
                 ; the reference is not blocking anymore, pass
                 ; the result
                 (ulm:ref-set! ret store)
                 ; and notify the waiting thread
                 (ulm:emit sig))))
        ; make him move with no respawn
        (gone (ulm:safe-migrate-to dest 0 #f name)))
  ; first check whether he's gone, if this throws, we let the exception
  ; be thrown, the agent won't respawn
  (ulm:await gone)
  ; now await its return
  (ulm:await sig)
  ; and return the thunk's return value
  (ulm:unref ret)))
```

Figure 3.22: Remote procedure call with a return value

```
<mixin-type-definition> ==> (define-mixin (<type-name> <argument-name> ...)
                              <mixin-spec> ...)
<mixin-spec>            ==> (inherit <mixin-type-expr> <expr> ...)
                            | (var <name> <expr>)
                            | (meth (<name> <super-name> <this-name>
                                            <argument-name> ...)
                               <expr> ...)
                            | (rename <old-name> <new-name>)
                            | (without <var-or-meth-name>)
<*name>                 ==> <identifier>
<mixin-type-expr>       ==> <expr>
```

Figure 3.23: Mixin definition

structures (based on SRFI-9) and provides an object system with single inheritance, introspection and method polymorphism. STklos [Gal] provides an object system based on CLOS (Common Lisp Object System) [Ste84] which supports multiple inheritance and a meta object protocol. PLT Scheme [Fla05] provides an object system based on classes, mixins and traits [FKF98] [FFF06].
ULM was designed with Boudol's mixins in mind [Bou01] as its object system, for which Zimmer made a first implementation [Zim04]. These mixins are a type of object system with multiple and parametrised inheritance: with mixins it is possible to define a mixin type which inherits from another mixin type given at runtime, and to add, remove or rename inherited methods or fields from such an inherited mixin type. It has been our work to integrate mixins in ULM, find an appropriate implementation and provide an introspection mechanism for runtime type checking.

Syntax

Mixin types are defined using the `define-mixin` form defined in Figure 3.23.
A mixin type definition introduces a new mixin type bound to the specified type name, which is a global identifier. The mixin type can be parametrised by the specified argument names, which will be bound and visible in the scope of the mixin upon instantiation. Mixin inheritance is introduced by the `inherit` clause, which takes an expression which must yield a mixin type and the values of the arguments expected by this mixin type.
Mixin fields consist in variables and methods. Variables are declared using the `var` clause, which takes the name of the variable to bind in the mixin and the value to bind it to. Methods are declared using the `meth` clause, which is similar to the `(define (<name> <argument-name> ...) <expr> ...)` scheme form. Two extra variables are bound by the system in the body of each mixin method: `super`, which is bound to the mixin's super type instance, and `this` which is bound to the current mixin's instance.
Since a mixin type can inherit from several mixin types, the super type refers to the type of mixin whose definition consists exclusively of the mixin clauses located before the method: anything specified below that method will not be visible in that method's super type instance.

```
(define p1 (new point 0 2))
p1.x
=> 0
(set! p1.x 3)
p1.x
=> 3
```

Figure 3.24: Using the point mixin

Fields (methods or variables) can be removed from the mixin type using the `without` clause, and renamed using the `rename` clause. This is key to solving the traditional problems of multiple-inheritance: method or field resolution. A field or method access always refers to the latest definition of the field or method with that name. Every redefinition *hides* the previous definition, which remains accessible only through the super type. If it is desirable to keep two fields sharing a single name obtained through multiple-inheritance, the first inherited field can be renamed, so that the second inherited field will not hide it.
Mixin types can be instantiated using the `(new <mixin-type-expr> <expr> ...)` syntax, which takes the mixin type as first argument, and the values for the mixin type's arguments next.
Mixin fields can be accessed using the traditional object system's "." notation:
`<mixin-instance-expr>.<field-name>`
This notation can also be used for setting mixin variables:
`(set! <mixin-instance-expr>.<variable-name> <expr>)`

Examples

Here is how one would create the typical Point type using mixins:

```
(define-mixin (point x-init y-init)
 (var x x-init)
 (var y y-init))
```

This example defines the mixin type point, with two variables x and y, whose initial values are arguments of the mixin type. Figure 3.24 shows how such a mixin instance would be created and used.
Figure 3.25 shows how we could extend our point mixin type by creating a resettable point.
Mixins enable us to build entirely new and adaptable synchronisation mechanisms on top of ULM signals. For example, it is possible to build a type of signal which can be *unemitted* during the instant[11]. Traditionally, we call *event* any type of synchronisation object based on ULM signals. Figure 3.26 shows how we would define the `disappearing-signal` type of event, which can be emitted, awaited and unemitted.
As described in an unpublished paper by Boudol, we can create a *valued* event, which can have a value associated with its emission. It is shown in Figure 3.27.
We can see from this valued event that only the last value emitted with `set` within an instant will be returned by `get`. Also note that it is reset at each new instant since its

[11] A blasphemy in the synchronous world, and admittedly not a very useful one.

```
(define-mixin (resettable-point x-init y-init)
 ; inherit from point and pass the arguments along
 (inherit point x-init y-init)
 ; the reset function
 (meth (reset)
  ; reset the x and y variables to their initial values
  (set! this.x x-init)
  (set! this.y y-init)))

(define p2 (new resettable-point 0 2))
p2.x
=> 0
(set! p2.x 3)
p2.x
=> 3
(p2.reset)
p2.x
=> 0
```

Figure 3.25: Extending point mixin

```
(define-mixin (disappearing-signal)
 (var signal (ulm:signal))
 ; this makes this event unemitted
 (meth (unemit)
  ; simply make a new signal
  (set! this.signal (ulm:signal)))
 ; emit this signal
 (meth (emit)
  (ulm:emit this.signal))
 ; wait for this event
 (meth (await)
  (ulm:await this.signal))
 ; tell us if this event has been emitted
 ; during the current instant
 (meth (present?)
  (ulm:present this.signal)))

(define p2 (new disappearing-signal))
(p2.emit)
(p2.present?)
=> #t
(p2.unemit)
(p2.present?)
=> #f
```

Figure 3.26: A dissapearing signal mixin

```
(define-mixin (valued-event)
 ; our synchronisation means
 (var signal (ulm:signal))
 ; this holds the associated value
 (var content #f)
 ; emits this event with a value
 (meth (set v)
  ; notify the emission
  (ulm:emit this.signal)
  ; and save the value
  (set! this.content v))
 ; gets the event value
 (meth (get)
  ; await a value
  (ulm:await this.signal)
  ; return it
  this.content))
```

Figure 3.27: A valued event mixin

`signal` variable gets reset too. So this event will start each instant with no value. If we want for the latest value to be preserved across instants, we could write it as in Figure 3.28.

As a last example, we will show how to build a multi-valued event mixin. This time we want each value emitted on the event to be stored by the event, and getting the value will yield the list of all values emitted during the current instant. Since the number of emission per instant can only be determined at the end of instant, getting the list of values emitted during an instant yields those values at the next instant.

This multi-valued event mixin overrides the `set` and `get` methods of the `event` mixin, and adds an `await` method, which first waits for the emission of the event before calling `get`.

This mixin still uses a signal for event emission triggering and waiting, and the `content` variable to store the list of values per instant. But this time, it contains a pair whose `car` contains the current instant, and whose `cdr` contains the list of values emitted during this instant. When we want to get the values emitted for the current instant with `get`, we start by installing the `content` variable by calling the `install-content` private method. This method will initialise the `content` variable for the current instant if it was not already done. To differentiate instants, we use the `ulm:instant` function which returns a unique number for each different instant. We then have to call `ulm:present` to see if this event's signal will be emitted during this instant. If it was emitted we skip to the next instant, otherwise `ulm:present` will already have returned at the next instant. Because we always end up at the next instant, we have to save the `content` variable before we skip instant, because it is only meaningful for the current instant. At the next instant we can return the list of values which were put in our `content` value via side-effect in `set`.

The `set` method consists in installing the `content` variable too, then modifying its `cdr` to augment it with the new value. The `await` method simply awaits the event's signal before calling `get` so it should not return an empty list of values, but can wait several instants (or infinitely) for this event to be emitted. The full mixin is shown in Figure 3.29.

```
(define-mixin (resilient-valued-event)
 ; inherit from valued-event
 (inherit valued-event)
 ; add a boolean variable to remember emission
 (var emitted #f)
 ; override set
 (meth (set v)
  ; remember emission
  (set! this.emitted #t)
  ; call the super-method
  (super.set v))
 ; override get
 (meth (get)
  (if this.emitted
   ; return the latest value
   this.content
   ; or await emission
   (super.get))))
```

Figure 3.28: A resilient valued event mixin

Introspection

ULM mixins solve the problem of having user data structures, but the typing of such mixins is a bit complex: Boudol's mixins have a static typing based on the idea that two mixins have the same type if they share the same fields and each has the same type, or a mixin A is a subtype of mixin B is A has at least the same fields (with the same types) as B.

While this would be essential in a statically typed language, in Scheme it is not enough: traditional scheme data types have some sort of typing information available at runtime, whether a simple type predicate (such as `pair?`) or more complex forms of introspection.

Our contribution in this mixin model is the addition of first-class mixin types. While Boudol's mixin types are defined as combinations of lambdas, we add a layer formally reifying mixin types in the same way Java classes are reified. Naturally, a mixin type is also a mixin instance, and defining a mixin type (with `define-mixin`) defines a global variable containing the mixin type instance.

The mixin type's type is called mixin-type, and if we had a `define-mixin-interface` form similar to `define-mixin` but without the variables' initial values and methods' body (which are not relevant here), its interface would be as shown in Figure 3.30.

Mixin types have the list of types inherited by such a type, its name, the list of the type's fields, its constructor and a method that looks up a field by name. Field descriptors are also mixins: mixin variables are described by the mixin-variable mixin type, and their methods by mixin-method defined as shown in Figure 3.31.

In order to have some hierarchy, a base mixin type named *mixin* is defined, which contains one variable containing the mixin instance's type. All mixins inherit from this base type:

```
; A multi-valued event mixin
(define-mixin (multi-valued-event)
 (inherit valued-event)
 ; private method which clears every past emission values
 ; and sets up a values list for this instant, then returns it
 (meth (install-content)
  (unless (= (ulm:instant) (car this.content))
   (set! this.content (cons (ulm:instant) '())))
  this.content)
 ; returns at the next instant the list of values emitted at
 ; this instant
 (meth (get)
  ; save the list
  (let ((content (this.install-content)))
   ; this returns #t during this instant, or #f at the next
   (if (ulm:present this.signal)
    (begin
     ; await the next instant before returning the values
     (ulm:pause)
     ; return all the values
     (cdr content))
    (begin
     ; no value
     '()))))
 ; returns the list of values emitted at the next instant of
 ; the first emission
 (meth (await)
  (ulm:await this.signal)
  (this.get))
 ; emit a value
 (meth (set val)
  (let ((content (this.install-content)))
   (set-cdr! content (cons val (cdr content)))
   (ulm:emit this.signal)))
 )
```

Figure 3.29: Event with multiple values

```
(define-mixin-interface mixin-type
 ; the list of inherited types
 (var types)
 ; the name of this mixin type
 (var name)
 ; the fields of this mixin type (variables and methods)
 (var fields)
 ; the constructor which creates such mixin instances
 (var constructor)
 ; method which gets a mixin field descriptor by name
 (meth (get-field name)))
```

Figure 3.30: The `mixin-type` mixin

```
(define-mixin-interface mixin-variable
 ; this variable's name
 (var name))

(define-mixin-interface mixin-method
 ; this method's name
 (var name)
 ; this method's arity
 (var arity)
 ; #t if this method's arity is fixed, #f if it can
 ; take more arguments
 (var fixed))
```

Figure 3.31: Mixin field introspection

```
(define-mixin-interface mixin
 ; this mixin's type instance
 (var type))
```

Since every mixin is guaranteed to inherit from the mixin type, every mixin instance has a `type` field. This includes all the super instances available in methods. When defining a mixin, the super instances in its methods cannot have the final full mixin's type, because super instances only see the mixin clauses located above the method. For such super instances we have created a system of anonymous mixin types. To understand this, let us look at the types of the mixins in Figure 3.32.
For each new method introduced within a mixin, a new type is created for the method's super instance. In theory, a new type should be created for every new mixin clause, but since instances of those types are only visible in methods, it makes sense to simplify the hierarchy by aggregating the super types located between methods. The output of our example is shown in Figure 3.33.
We can see that for each new method, an anonymous super type is created, which has all the fields defined above the method, and whose super types includes the the previously inherited types. Each new anonymous super type inherits from the previous anonymous super type, and the final type (`foo` here) inherits from the last anonymous super type.
With this system it is possible to describe every mixin type at runtime, including list its variables, methods (since accessing them is dynamic, accessors and modifiers are not necessary in the reflection system), get its constructor, name and list of inherited mixin types.

Mixin type compatibility

In Scheme every value can be determined to be of a certain type with a predicate per data type. The primitive `mixin?` returns `#t` if the given value is a mixin instance, but this is not enough to know precisely what type of mixin it is. We have to explain how we can write a predicate which returns `#t` if a given value is of a given mixin type.
Boudol's static typing system for its mixins consists in comparing two mixin types to check that they have the same interface: same variables and functions in the resulting instances. This is necessary because inheriting from a mixin type is not enough to be a subtype of it if we remove or rename some of its fields. Take the following example of crippled-point which inherits from the point mixin type we defined earlier:

```
(define-mixin (crippled-point)
 ; inherit from the point mixin
 (inherit point 0 0)
 ; but remove its x variable
 (without x))
```

The crippled-point mixin inherits from point, yet is missing its x field, having only the y field. Clearly it is not a subtype of point because it cannot be used as such: any access to a crippled-point instance as a point will cause an error when accessing its missing x field. This is why Boudol's definition of mixin types and subtypes specify that a subtype must have all the fields of the supertype, each with the same type. But this is not enough, because it is easy to create two mixin types with the same fields while they are clearly not of the same type:

```
; This function shows the mixin's type name, its fields and super-types
(define (dump-type m)
 (print "Mixin type name: " m.type.name)
 (display " Mixin field names: ")
 (for-each (lambda (t)
             (display t.name " "))
  m.type.fields)
 (newline)
 (display " Mixin super types names: ")
 (for-each (lambda (t)
             (display t.name " "))
  m.type.types)
 (newline))

; We define two mixins with no field
(define-mixin (A))
(define-mixin (B))
; And a mixin inheriting from A and B
; with accessors to get the super instances
(define-mixin (foo)
 (meth (get-super1) super)
 (inherit A)
 (meth (get-super2) super)
 (inherit B)
 (meth (get-super3) super))

; Instantiate our mixin
(define f (new foo))
; Dump all super instances' types
(dump-type (f.get-super1))
(dump-type (f.get-super2))
(dump-type (f.get-super3))
; And the instance's type
(dump-type f)
```

Figure 3.32: Mixin introspection

```
Mixin type name: $anonymous-mixin-foo1496
 Mixin field names: type
 Mixin super types names: mixin
Mixin type name: $anonymous-mixin-foo1497
 Mixin field names: type get-super1
 Mixin super types names: A $anonymous-mixin-foo1496
Mixin type name: $anonymous-mixin-foo1498
 Mixin field names: type get-super2 get-super1
 Mixin super types names: B $anonymous-mixin-foo1497
Mixin type name: foo
 Mixin field names: get-super3 get-super1 get-super2 type
 Mixin super types names: $anonymous-mixin-foo1498
```

Figure 3.33: Mixin introspection output

```
(define-mixin-interface hamburger
 (meth (slice x)))

(define-mixin-interface resizable-array
 (meth (slice x)))
```

Clearly the two have nothing in common but their interface is the same, yet in our view they should not be of the same type. In order to clarify this, we state that mixin A has the type B if its `type` field is B, is a subtype of B if and only if all the following conditions are met:

1. A inherits from B directly or indirectly.
2. A contains all of B's variables.
3. A contains all of B's methods.
4. Those methods have the same arity in A and B.

We believe the addition of the inheritance constraint enforces the fact that subtyping has to be conscious to the programmer, while the field compatibility checks are here to make sure the subtyping was not destroyed. It is still possible for a mixin to remove and replace an inherited method, but as long as the replaced method has the same arity, we believe it is a natural method override done consciously by the programmer, and should not change the mixin type's compatibility with its super type.
The `isa` function takes an object and a mixin type as arguments and returns true if the given object is of the same type or is a subtype of the given mixin type. We believe this is enough to build predicates for all mixin types.

3.2.12 Modules

Our implementation of ULM supports code modularisation using modules. Every ULM file has to be composed of exactly one module, whose name has to be unique. A module

```
<module-definition> ==> (module <module-name>
                          <module-spec> ...)
<module-spec>       ==> (import <module-name> ...)
                      | (export <export-spec> ...)
                      | (extern <extern-spec> ...)
                      | (main <main-name>)
<export-spec>       ==> <identifier>
                      | (<identifier> <argument-name> ...)
                      | (macro (<identifier> <argument-name> ...))
<extern-spec>       ==> (<module-name> (<native-binding> ...)
                           <export-spec> ...)
<native-binding>    ==> (<native-type> . <native-module>)
```

Figure 3.34: Module definition

consists of a name, a list of imported modules, a list of exported variables, a list of *extern* native bindings, the name of the main function and a toplevel which consists of the module's body.

Modules are described using the `module` clause at the top of each module, defined as shown in Figure 3.34.

The `import` clause lists the name of modules to be imported in this module: all the symbols exported by those symbols will be visible from this module's body. The `export` clause lists the name of variables exported from this module. Each exported symbol can have three forms: a symbol name to export a variable whose type is unknown, a function prototype to export a function with an explicit prototype, or a macro prototype. All exported symbols have to be defined within the module, with an appropriate type, and are imported read-only by other modules.

The `extern` clause imports symbols from non-ULM sources, so-called *native* bindings. As we will see in 4.1 ULM runs on several backends, so it is possible to specify which native module (backend dependent) provides a given native module. As native modules are not written in ULM, they do not provide the list of exported symbols or their type or prototype, so they are specified for each imported native module. The native modules are explained in detail in Chapter 6.

The `main` clause specifies the function which should be invoked if this module is the main ULM module being executed, after the toplevel. With modules a symbol is looked up in the given order: local variable, global variable, imported or native variable.

Modules are looked up, loaded and initialised at runtime (and compile time actually) *just in time*, which means that if the symbol X is exported by the module B, which is imported by the module A, the module B's toplevel will be executed at the first attempted read of the variable X. This clears most circular module dependencies and gets rid of unused module initialisation costs.

Let us illustrate how we can create interdependent modules without any initialisation problem. Figure 3.35 shows a module `foo` which exports the function `f` and the mixins `A` and `B`.

The module `bar` imported by the `foo` module is then shown in Figure 3.36. It exports the

```
(module foo
 (import bar)
 (export A B f))

(print "Loading module foo")

(define (f x)
 (if (> x 0)
  (g (- x 1))
  x))

(define-mixin (A)
 (var f #f)
 (meth (g x)
  (print "A.g " x)
  x))

(define-mixin (B)
 (inherit C)
 (meth (g x)
  (print "B.g " x)
  (super.g x)))

(print "Module foo loaded")
```

Figure 3.35: Interdependent modules: foo

```
(module bar
 (import foo)
 (export C g))

(print "Loading module bar")

(define (g x)
 (if (> x 0)
  (f (- x 1))
  x))

(define-mixin (C)
 (inherit A)
 (meth (g x)
  (print "C.g " x)
  (super.g x)))

(print "Module bar loaded")
```

Figure 3.36: Interdependent modules: bar

function `g` and the mixin `C`.

We can see that the functions `f` and `g` are interdependent, while the `B` mixin inherits from `C` which inherits from `A`, inheriting across interdependent modules. However, since both modules' toplevel only contain definitions, execution of either module will not trigger the loading of the other. It is even possible to put non-definition expressions requiring both modules in the toplevel of either one (not both) module's toplevel. For example, by appending this to either module:

```
(define t (new B))
(t.g 2)
```

Will result in both modules being loaded prior to `B`'s instantiation:

```
Loading module foo
Module foo loaded
Loading module bar
Module bar loaded
B.g 2
C.g 2
A.g 2
```

This works because in both cases, the other module will be fully loaded once we start using its exported values. Of course, our module system has no solution in the case of two variables needing each other's value for their definition, but this is the same with or without modules.

3.3 Semantics

As we hope to have demonstrated by now, ULM provides a deterministic scheduling. In this section we will describe ULM primitives as they have been integrated in Scheme. For that we will use denotational semantics as traditionally used for Scheme. These semantics' notations are based on Bigloo's Fair Threads semantics [SBS04], based in turn on Queinnec's L.I.S.P. book [Que96] and on R^5RS's semantics.

3.3.1 Notation

In order to better apprehend these formal semantics (in all their Greek-friendly notation beauty) we will start by describing how evaluating ULM code works.
Expressions are represented by the symbol π and are evaluated by the function ε. Since all evaluation is done in a ULM thread, the evaluation function needs arguments which are thread-specific, such as the thread's current environment ρ, its current continuation κ, its context ζ and store θ.
The thread's context consists in the list of current suspension (`when`), and preemption (`watch`) clauses, as well as clauses introduced by strong preemption (`escape`) and protection (`protect`). The first clause in the context is the bottommost clause while the last one is the topmost clause. There are two more clauses which are *terminal* because they can only appear at the beginning of the context: those introduced by waiting on a reference (`ref`), and waiting using `ulm:await` (which is represented as `when` too, since it is derived from it). You can view this list of clauses as ULM's dynamic context.
The thread's current store represents either the current site's store, or a store inherited by an agent.
Aside from those arguments belonging to the current thread, two more arguments are needed by the evaluator, which are given by the scheduler: the set of signals emitted at the current instant σ and the set of stores present on the site during this instant Θ. These two arguments are global (on one site, of course), and thus can be modified and visible by all threads, so they are passed to the continuations with the evaluated value.
The evaluator function returns a list consisting of the thread state τ (composed of its scheduling status, continuation, context and store), the two globals (signals and stores), a list of new threads (T), and a list of migrations (M) to trigger. This is of course used by the scheduler, which we will describe later in 3.3.3.
The list of abbreviations used in these semantics is shown in Figure 3.37.
The conditional evaluation is denoted with "$\pi_1 \rightarrow \pi_2, \pi_3$", with π_2 or π_3 evaluated if π_1 is true or false respectively. The list concatenation is represented with "§", the list difference with "\", obtaining the n^{th} element (starting at 1) with "$L\downarrow_n$", obtaining the list with the n^{th} first elements removed with "L†n". The tuple reference to its n^{th} member (starting at 1) is denoted with "$T\downarrow_n$". The boolean conjunction is denoted with "∧", the disjunction with "∨" and the complement with "¬". Evaluator values are converted to ULM values with the function "inValue", while ULM values are converted to evaluator values of type t with the notation "$\varepsilon|_t$".

3.3.2 Evaluation

The evaluation function ε evaluates a Scheme expression in a non-suspensive context. The scheduler verifies that before evaluating an expression, and the evaluator does too before

		π		**Expr**
		ν		**Identifier**
		η		**String**
		π		**Expr**
		θ		**Store**
		ς		**Signal**
ε	$\in$	**Value**	$=$	**Fun** + **Int** + **Cons** + **String** + **Bool** + **Signal** + **Ref** + ...
φ	$\in$	**Fun**	$=$	**Value** $\times$ **Signals** $\times$ **Stores** $\times$ **Ctx** $\times$ **Store** $\times$ **Cont** $\rightarrow$ **Value**
κ	$\in$	**Cont**	$=$	**Value** $\times$ **Signals** $\times$ **Stores** $\rightarrow$ **Value**
ρ	$\in$	**Env**	$=$	**Ident** $\rightarrow$ **Value**
		Ref	$=$	**Value** $\times$ **Store**
σ	$\in$	**Signals**	$=$	**Signal***
Θ	$\in$	**Stores**	$=$	**Store***
ζ	$\in$	**Ctx**	$=$	**CClause***
ι	$\in$	**CClause**	$=$	**When** + **Watch** + **Ref** + **Protect** + **Escape**
		When	$=$	`when` $\times$ **Signal**
		Watch	$=$	`watch` $\times$ **Signal** $\times$ **Cont**
		Ref	$=$	`ref` $\times$ **Store**
		Protect	$=$	`protect` $\times$ **Cont** $\times$ **Signals** $\times$ **Stores** $\rightarrow$ **Value**
		Escape	$=$	`escape` $\times$ **Cont**
T	$\in$	**Threads**	$=$	**Thread***
τ	$\in$	**Thread**	$=$	**ThrdStatus** $\times$ **Cont** $\times$ **Ctx** $\times$ **Store**
		ThrdStatus	$=$	`pause` + `terminated` + `ready`
M	$\in$	**Migrations**	$=$	**Migration***
μ	$\in$	**Migration**	$=$	`to` $\times$ `String` $\times$ `Store` $\times$ `Int` $\times$ `Bool` $\times$ `SigOrFalse`
		SigOrFalse	$=$	**Signal** + **False**
ω	$\in$	**ThrdResult**	$=$	**Thread** $\times$ **Signals** $\times$ **Stores** $\times$ **Thread** $\times$ **Migration**
Ω	$\in$	**SchdResult**	$=$	**Threads** $\times$ **Signals** $\times$ **Stores** $\times$ **Threads** $\times$ **Migrations**
ψ	$\in$	**SigsAndThrds**	$=$	**Signals** $\times$ **Threads**
		$\mathcal{E}$	:	**Expr** $\rightarrow$ **Env** $\times$ **Signals** $\times$ **Stores** $\times$ **Ctx** $\times$ **Store** $\times$ **Cont** $\rightarrow$ **ThrdResult**
		$\mathcal{S}$	:	**Thread** $\rightarrow$ **Signals** $\times$ **Stores** $\times$ **Threads** $\times$ **Migrations** $\rightarrow$ **SchdResult**

Figure 3.37: Abbreviation used in the semantics

entering a potentially suspensive context.

Standard Scheme

For a variable access, its value is fetched from the environment and passed to the continuation with the signals and stores unchanged.

$$\mathcal{E}[\![\nu]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\kappa\ (\rho\ \nu)\ \sigma\ \Theta)$$

For a quotation, the expression's source code is passed to the continuation without evaluation. The signals and stores are unchanged.

$$\mathcal{E}[\![(\texttt{quote}\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\kappa\ \pi\ \sigma\ \Theta)$$

For a variable affectation we evaluate π, then assign that value to the variable ν. *assign* modifies the environment, and throws an error if the variable is unknown.

$$\mathcal{E}[\![(\texttt{set!}\ \nu\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\kappa\ (\mathit{assign}\ \nu\ \rho)\ \sigma_1\ \Theta_1))$$

The conditional involves evaluating π, then if it evaluates to anything but the false value, evaluate π_1, otherwise evaluate π_2. *boolify* returns false for the scheme value `#f`, true otherwise.

$$\mathcal{E}[\![(\texttt{if}\ \pi\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\mathit{boolify}\ \varepsilon) \rightarrow (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \kappa), (\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \kappa))$$

For the sequence, we evaluate the first expression π_1 before dropping its value and evaluating the second expression π_2. Of course, sequences of more than two expressions can be composed as `(begin` π_1 π_2`)`.

$$\mathcal{E}[\![(\texttt{begin}\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \kappa))$$

The lambda creation involves capturing the current environment ρ and saving the name ν and body π. To that end we create a function which will take the value we will use for ν (ε), the globals, the dynamic context in which the lambda will be called, and the continuation of the call. When that function will be applied it will evaluate π in the captured environment ρ augmented with ν bound to ε with the continuation of the caller. We then wrap this function as a scheme value and pass it to the continuation κ. Functions of different arities can be obtained by composition, and are thus not described.

$$\mathcal{E}[\![(\lambda\ (\nu)\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\kappa\ (\text{inValue}\ \lambda\varepsilon\sigma_1\Theta_1\zeta_1\theta_1\kappa_1.(\mathcal{E}[\![\pi]\!]\ \rho[\nu/\varepsilon]\ \sigma_1\ \Theta_1\ \zeta_1\ \theta_1\ \kappa_1))\ \sigma\ \Theta)$$

The last purely Scheme rule is the function call. We start by evaluating π_1 as the function

φ, then the argument π_2 as ε. Then we unwrap φ to the function we created in the abstraction rule, and call it by passing it ε, the globals, the dynamic environment and the current continuation.

$$\mathcal{E}[\![(\pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa =$$
$$(\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varphi\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \lambda\varepsilon\sigma_2\Theta_2.(\varphi|_{\text{Function}}\ \varepsilon\ \sigma_2\ \Theta_2\ \zeta\ \theta\ \kappa)))$$

With the protection block introduced by `unwind-protect` we see the first dynamic context clause: `protect`. `unwind-protect` involves evaluating π_1 into ε_1, then evaluating π_2 and calling the continuation with ε_1 when the evaluation of π_1 is not preempted. If it is preempted we have to evaluate π_2 before unwinding.

This is done by evaluating π_1 in a context augmented with a `protect` clause containing a protector function. Should that evaluation be preempted, that function will be invoked (in *unwind-to* which we describe later) with a new continuation κ_3 and the two globals as arguments. It will then evaluate π_2 with the continuation κ_3, which will likely keep on unwinding. In both places where we evaluate π_2 the context is stripped of the protection clause.

$$\mathcal{E}[\![(\texttt{unwind-protect}\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa =$$
$$(\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \langle\texttt{protect}\ \lambda\kappa_3\sigma_3\Theta_3.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_3\ \Theta_3\ \zeta\ \theta\ \kappa_3)\rangle\S\zeta\ \theta$$
$$\lambda\varepsilon_1\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \lambda\varepsilon_2\sigma_2\Theta_2.(\kappa\ \varepsilon_1\ \sigma_2\ \Theta_2)))$$

The strong preemption primitive `bind-exit` evaluates π with a new local variable ν containing an escaper function which unwinds to the continuation of `bind-exit`. To that end we evaluate π with an environment augmented with ν bound to that escaper function wrapped in a scheme value. When called, that function will invoke *unwind-to* with the list of protection clauses contained in the dynamic extent between the call to the escaper function and the call to `bind-exit`, as well as a continuation which involves invoking `bind-exit`'s continuation κ with the value passed as argument to our escaper function.

$$\mathcal{E}[\![(\texttt{bind-exit}\ (\nu)\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho[\nu/(\text{inValue}\ \lambda\varepsilon\sigma_1\Theta_1\zeta_1\theta_1\kappa_1.(\mathit{unwind\text{-}to}\ (\mathit{filter\ protect?}\ \zeta_1\backslash\zeta)$$
$$\lambda\varepsilon_2\sigma_2\Theta_2.(\kappa\ \varepsilon\ \sigma_2\ \Theta_2)\sigma_1))]\ \sigma\ \Theta\ \zeta\ \theta\ \kappa)$$

The *unwind-to* evaluator function takes a list of protection clauses P, a final continuation κ as well as the globals. If P is empty it invokes the final continuation κ, otherwise it invokes the first clauses' protector function with a continuation calling *unwind-to* with the next protection clauses. Its effect is thus to call all protection clauses' functions contained in P and then calling κ.

$$\mathit{unwind\text{-}to} = \lambda\text{P}\kappa\sigma\Theta.\text{P}{=}\emptyset \rightarrow (\kappa\ \mathit{unspecified}\ \sigma\ \Theta),$$
$$((\mathit{protect\text{-}func}\ \text{P}{\downarrow_1})\ \lambda\varepsilon\sigma_1\Theta_1.(\mathit{unwind\text{-}to}\ \text{P}{\dagger}1\ \kappa\ \sigma_1\ \Theta_1)\ \sigma\ \Theta)$$

ULM primitives

The first, and simplest ULM primitive is the signal creation, which involves only creating a new signal and passing it to the continuation κ.

$$\mathcal{E}[\![(\texttt{ulm:signal})]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\kappa\ (\textit{new-signal})\ \sigma\ \Theta)$$

Emitting a signal consists in evaluating π and adding it to the set of emitted signals σ_1.

$$\mathcal{E}[\![(\texttt{ulm:emit}\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\kappa\ \textit{unspecified}\ \sigma_1 \S \varepsilon\ \Theta_1))$$

Entering a when control block involves evaluating π_1 into a signal ε, then augmenting the context ζ into ζ_1 by adding a `when` clause with the signal ε. At this point the continuation is to evaluate π_2 with the new context ζ_1. We store this continuation in κ_w. Before we can invoke this continuation however, we test whether our signal ε is in the set of emitted signals σ_1. If it is, we invoke directly that continuation. If the signal hasn't been emitted, we have to yield to the scheduler, with the continuation κ_w and the new context ζ_1.

$$\mathcal{E}[\![(\texttt{ulm:when}\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.\underline{\texttt{let}}\ \zeta_1 = \langle\texttt{when}\ \varepsilon\rangle \S \zeta, \\ \kappa_\text{w} = \lambda\varepsilon_2\sigma_2\Theta_2.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_2\ \Theta_2\ \zeta_1\ \theta\ \kappa)\ \underline{\texttt{in}} \\ \varepsilon \in \sigma_1 \rightarrow (\kappa_\text{w}\ \textit{unspecified}\ \sigma_1\ \Theta_1), \langle\langle\texttt{ready}\ \kappa_\text{w}\ \zeta_1\ \theta\rangle, \sigma_1, \Theta_1, \emptyset, \emptyset\rangle)$$

Waiting for a signal with `ulm:await` is very similar to the suspension block, except that it is always *terminal*: no expression is evaluated in a new context. We start by evaluating π into the signal ε_s, then we create the continuation κ_a which should be evaluated when the signal ε_s will be emitted. That continuation consists in checking if the signal is a signal that was created for safe migration notification, and whether the migration in question has been marked as *failed* with the predicate *signal-failure?*. If the signal signaled a failed migration, we evaluate a call to a library function named `ulm:raise-migr-error` which is in charge of raising a migration error using the exception mechanism library described previously. Otherwise the continuation to `ulm:await` κ is called.
Like in the suspension primitive, before invoking κ_a we check whether the signal ε_s was emitted. If it was κ_a is invoked immediately. Otherwise, we yield to the scheduler with the context ζ augmented with a `when` clause with the signal ε_s. What is important to note here is that this new context is not used for evaluation anywhere (contrary to the suspension primitive): it is just used so that the scheduler knows when it can schedule us again.

$$\mathcal{E}[\![(\texttt{ulm:await}\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon_\text{s}\sigma_\text{s}\Theta_\text{s}.\underline{\texttt{let}}\ \kappa_\text{a} = \lambda\varepsilon_\text{a}\sigma_\text{a}\Theta_\text{a}.(\textit{signal-failure?}\ \varepsilon_\text{s}) \rightarrow \\ (\mathcal{E}[\![(\texttt{ulm:raise-migr-error})]\!]\ \rho\ \sigma_\text{a}\ \Theta_\text{a}\ \zeta\ \theta\ \kappa), (\kappa\ \textit{unspecified}\ \sigma_\text{a}\ \Theta_\text{a})\ \underline{\texttt{in}} \\ \varepsilon_\text{s} \in \sigma_\text{s} \rightarrow (\kappa_\text{a}\ \varepsilon_\text{s}\ \sigma_\text{s}\ \Theta_\text{s}), \langle\langle\texttt{ready}\ \kappa_\text{a}\ \langle\texttt{when}\ \varepsilon_\text{s}\rangle \S \zeta\ \theta\rangle, \sigma_\text{s}, \Theta_\text{s}, \emptyset, \emptyset\rangle)$$

The preemption primitive consists in evaluating π_1 into the signal ε, then evaluating the body π_2 with the context ζ augmented with a `watch` clause with the signal ε and the

`ulm:watch` continuation κ. The clause's continuation will be used at the end of the instant if the scheduler determines that the clause's signal has been emitted.

$$\mathcal{E}[\![(\texttt{ulm:watch}\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa =$$
$$(\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \langle\texttt{watch}\ \varepsilon\ \kappa\rangle\S\zeta\ \theta\ \kappa))$$

The `ulm:pause` function (normally derived with `ulm:when` and `ulm:watch`) has been made a primitive. It consists in yielding to the scheduler with the `pause` status which means that this thread should not be scheduled again until the next instant.

$$\mathcal{E}[\![(\texttt{ulm:pause})]\!]\rho\sigma\Theta\zeta\theta\kappa =$$
$$\langle\langle\texttt{pause}\ \kappa\ \zeta\ \theta\rangle, \sigma, \Theta, \emptyset, \emptyset\rangle$$

New threads inherit two things from their creator: the store θ and the context ζ. The context is inherited using the function *inherit-context* into the new context ζ_1. Then we yield to the scheduler and give it the new thread, whose continuation consists in evaluating π in the ζ_1 context with a continuation of *end-kont*.

$$\mathcal{E}[\![(\texttt{ulm:thread}\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa =$$
$$\underline{\texttt{let}}\ \zeta_1 = (\textit{inherit-context}\ \zeta)\ \underline{\texttt{in}}$$
$$\langle\langle\texttt{ready}\ \kappa\ \zeta\ \theta\rangle, \sigma, \Theta, \langle\texttt{ready}\ \lambda\varepsilon\sigma_1\Theta_1.(\mathcal{E}[\![\pi]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta_1\ \theta\ \textit{end-kont})\ \zeta_1\ \theta\rangle, \emptyset\rangle$$

The function *inherit-context* takes a context from a parent thread, and creates a new context to be used by a child thread. Since it is not possible to unwind outside the dynamic extent of a thread, all `protect` clauses are filtered out. For the same reason, all `watch` clauses' continuations are replaced by *end-kont* so that inherited preemption terminates the new thread. Suspension clauses on the other hand, are preserved *as-is*.

$$\textit{inherit-context} = \lambda\zeta.\zeta{=}\emptyset \rightarrow \emptyset, \underline{\texttt{let}}\ \iota = \zeta{\downarrow_1}, \zeta_1 = \zeta{\dagger}1\ \underline{\texttt{in}}$$
$$(\textit{when?}\ \iota) \rightarrow \iota\S(\textit{inherit-context}\ \zeta_1),$$
$$(\textit{watch?}\ \iota) \rightarrow \langle\texttt{watch}\ (\textit{watch-signal}\ \iota)\ \textit{end-kont}\rangle\S(\textit{inherit-context}\ \zeta_1),$$
$$(\textit{inherit-context}\ \zeta_1)$$

Every thread's final continuation is *end-kont* and consists only in yielding to the scheduler with a `terminated` status. Note that the thread's store is not removed from Θ because stores are present until the end of instant, so the scheduler will filter it out then.

$$\textit{end-kont} = \lambda\varepsilon\sigma\Theta.\langle\langle\texttt{terminated}\ \textit{end-kont}\ \emptyset\ \text{false}\rangle, \sigma, \Theta, \emptyset, \emptyset\rangle$$

Creating an agent differs in many ways from creating a thread. First we create a new name for the agent in η, then we create a new store for it in θ_1. We then yield to the scheduler with a continuation that returns the new name to the caller. We also give the scheduler that new agent, whose continuation consists in evaluating its body π with the current environment ρ augmented with the variable ν bound to the new name η, an empty context, the new store θ_1 and the final continuation *end-kont*.

$$\mathcal{E}[\![(\texttt{ulm:agent}\ (\nu)\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa =$$

$\underline{\texttt{let}}\ \eta = (\textit{new-agent-name}),\ \theta_1 = (\textit{new-store})\ \underline{\texttt{in}}$
$\quad \langle\langle \texttt{ready}\ \lambda\varepsilon_1\sigma_1\Theta_1.(\kappa\ \eta\ \sigma_1\ \Theta_1)\ \zeta\ \theta\rangle,\ \sigma,\ \theta_1 \S \Theta,$
$\quad \langle \texttt{ready}\ \lambda\varepsilon_2\sigma_2\Theta_2.(\mathcal{E}[\![\pi]\!]\ \rho[\nu/\eta]\ \sigma_2\ \Theta_2\ \emptyset\ \theta_1\ \textit{end-kont})\ \emptyset\ \theta_1\rangle,\ \emptyset\rangle$

We create references by evaluating its initial value π into ε, then passing our continuation a new reference with that value in the current store θ.

$\mathcal{E}[\![(\texttt{ulm:ref}\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa =$
$\quad (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\kappa\ (\textit{new-ref}\ \theta\ \varepsilon)\ \sigma_1\ \Theta_1))$

For dereferencing, we evaluate π into the reference ε, then we check whether its store is currently present on the site by checking if it is in Θ_1. If yes, we pass the reference's value to our continuation. If the reference's store is not present, we yield to the scheduler with a continuation of returning the reference's content, and a context augmented with a `ref` clause on the reference's store. Just like for `ulm:await` that clause is *terminal* and no evaluation will take place using it, it is there only for the scheduler to determine when to schedule the thread back.

$\mathcal{E}[\![(\texttt{ulm:unref}\ \pi)]\!]\rho\sigma\Theta\zeta\theta\kappa =$
$\quad (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon\sigma_1\Theta_1.(\textit{ref-store}\ \varepsilon)\in\Theta_1 \rightarrow (\kappa\ (\textit{ref-value}\ \varepsilon)\ \sigma_1\ \Theta_1),$
$\qquad \langle\langle \texttt{ready}\ \lambda\varepsilon_2\sigma_2\Theta_2.(\kappa\ (\textit{ref-value}\ \varepsilon)\ \sigma_2\ \Theta_2)\ \langle \texttt{ref}\ (\textit{ref-store}\ \varepsilon)\rangle \S\zeta\ \theta\rangle,\ \sigma_1,\ \Theta_1,\ \emptyset,\ \emptyset\rangle)$

Affecting a reference is very similar: we evaluate the reference π_1 into ε_1, the new value π_2 in ε_2, then if the reference's store is in Θ_2, change reference's value. Otherwise yield to the scheduler with a new `ref` context on the reference's store and a continuation that will change the reference's value when the store is there.

$\mathcal{E}[\![(\texttt{ulm:ref-set!}\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa =$
$\quad (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon_1\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \lambda\varepsilon_2\sigma_2\Theta_2.(\textit{ref-store}\ \varepsilon_1)\in\Theta_2 \rightarrow$
$\qquad (\kappa\ (\textit{ref-value-set!}\ \varepsilon_1\ \varepsilon_2)\ \sigma_1\ \Theta_1),$
$\qquad \langle\langle \texttt{ready}\ \lambda\varepsilon_3\sigma_3\Theta_3.(\kappa\ (\textit{ref-value-set!}\ \varepsilon_1\ \varepsilon_2)\ \sigma_3\ \Theta_3)\ \langle \texttt{ref}\ \varepsilon_1\rangle \S\zeta\ \theta\rangle,\ \sigma_2,\ \Theta_2,\ \emptyset,\ \emptyset\rangle))$

Migration of the agent π_2 to the site π_1 is done by evaluating those to ε_2 and ε_1 respectively, then yielding to the scheduler with a migration order. This migration order contains the destination site ε_1, the agent's store we want to migrate (obtained from the 1:1 mapping between agent names and stores), and a set of default values given only for safe migration: we want no timeout (value of 0), no respawn and no signal emitted for notification.

$\mathcal{E}[\![(\texttt{ulm:migrate-to}\ \pi_1\ \pi_2)]\!]\rho\sigma\Theta\zeta\theta\kappa =$
$\quad (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon_1\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta$
$\qquad \lambda\varepsilon_2\sigma_2\Theta_2.\langle\langle \texttt{ready}\ \kappa\ \zeta\ \theta\rangle,\ \sigma_2,\ \Theta_2,\ \emptyset,\ \langle \texttt{to}\ \varepsilon_1\ (\textit{name->store}\ \varepsilon_2)\ 0\ \text{false false}\rangle\rangle))$

Subjective migration is very similar except the current store is used instead of obtaining it from a argument.

$\mathcal{E}[\![(\texttt{ulm:migrate-to}\ \pi_1)]\!]\rho\sigma\Theta\zeta\theta\kappa =$
$\quad (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon_1\sigma_1\Theta_1.\langle\langle \texttt{ready}\ \kappa\ \zeta\ \theta\rangle,\ \sigma_1,\ \Theta_1,\ \emptyset,\ \langle \texttt{to}\ \varepsilon_1\ \theta\ 0\ \text{false false}\rangle\rangle)$

Safe objective migration of the agent π_4 to the site π_1 with a timeout of π_2 and a respawn of π_3 is done by evaluating them respectively to ε_4, ε_1, ε_2 and ε_3. We create a new signal which we return to our continuation, and yield to the scheduler to give the order of migration to the target site of the corresponding store, with the timeout, the respawn value and the signal to emit when the migration is done.

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{ulm:safe-migrate-to}\ \pi_1\ \pi_2\ \pi_3\ \pi_4)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\
&\quad (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon_1\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \lambda\varepsilon_2\sigma_2\Theta_2.(\mathcal{E}[\![\pi_3]\!]\ \rho\ \sigma_2\ \Theta_2\ \zeta\ \theta \\
&\qquad \lambda\varepsilon_3\sigma_3\Theta_3.(\mathcal{E}[\![\pi_4]\!]\ \rho\ \sigma_3\ \Theta_3\ \zeta\ \theta\ \lambda\varepsilon_4\sigma_4\Theta_4.\underline{\texttt{let}}\ \varsigma = (\textit{new-signal})\ \underline{\texttt{in}} \\
&\qquad \langle\langle\texttt{ready}\ \lambda\varepsilon_5\sigma_5\Theta_5.(\kappa\ \varsigma\ \sigma_5\ \Theta_5)\ \zeta\ \theta\rangle, \sigma_4, \Theta_4, \emptyset, \\
&\qquad \langle\texttt{to}\ \varepsilon_1\ (\textit{name->store}\ \varepsilon_4)\ \varepsilon_2|_{\text{Number}}\ (\textit{boolify}\ \varepsilon_3)\ \varsigma\rangle\rangle)))))
\end{aligned}$$

The safe subjective migration is the same except for the current store being used.

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{ulm:safe-migrate-to}\ \pi_1\ \pi_2\ \pi_3)]\!]\rho\sigma\Theta\zeta\theta\kappa = \\
&\quad (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \Theta\ \zeta\ \theta\ \lambda\varepsilon_1\sigma_1\Theta_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \Theta_1\ \zeta\ \theta\ \lambda\varepsilon_2\sigma_2\Theta_2.(\mathcal{E}[\![\pi_3]\!]\ \rho\ \sigma_2\ \Theta_2\ \zeta\ \theta \\
&\qquad \lambda\varepsilon_3\sigma_3\Theta_3.\underline{\texttt{let}}\ \varsigma = (\textit{new-signal})\ \underline{\texttt{in}} \\
&\qquad \langle\langle\texttt{ready}\ \lambda\varepsilon_4\sigma_4\Theta_4.(\kappa\ \varsigma\ \sigma_4\ \Theta_4)\ \zeta\ \theta\rangle, \sigma_3, \Theta_3, \emptyset, \\
&\qquad \langle\texttt{to}\ \varepsilon_1\ \theta\ \varepsilon_2|_{\text{Number}}\ (\textit{boolify}\ \varepsilon_3)\ \varsigma\rangle\rangle))))
\end{aligned}$$

3.3.3 Scheduling

The scheduling function skips over threads marked as waiting for the next instant with a `pause` status.

$$\begin{aligned}
&\mathcal{S}[\![(\texttt{pause}\ \kappa\ \zeta\ \theta)]\!]\sigma\Theta\mathrm{T}\ \mathrm{M} = \\
&\quad \langle\langle\texttt{pause}\ \kappa\ \zeta\ \theta\rangle, \sigma, \Theta, \mathrm{T}, \mathrm{M}\rangle
\end{aligned}$$

Terminated threads with a status of `terminated` are also skipped.

$$\begin{aligned}
&\mathcal{S}[\![(\texttt{terminated}\ \kappa\ \zeta\ \theta)]\!]\sigma\Theta\mathrm{T}\ \mathrm{M} = \\
&\quad \langle\langle\texttt{terminated}\rangle, \sigma, \Theta, \mathrm{T}, \mathrm{M}\rangle
\end{aligned}$$

Threads marked as `ready` have their context tested for suspension using the *suspensive?* function, which returns true if the given context is suspensive for the given sets of signals and stores. If the thread should be suspended, it is skipped by the scheduler. Otherwise its continuation is invoked and we reschedule the thread with the values it returned: its new state, the new sets of signals and stores, any new thread we append to T, or migration order we append to M. Rescheduling the returned thread until it is terminated or waiting is done because thread creation or migration orders are not suspensive, and the thread should therefore keep on being scheduled.

$$\begin{aligned}
&\mathcal{S}[\![(\texttt{ready}\ \kappa\ \zeta\ \theta)]\!]\sigma\Theta\mathrm{T}\ \mathrm{M} = \\
&\quad (\textit{suspensive?}\ \zeta\ \sigma\ \Theta) \rightarrow \langle\langle\texttt{ready}\ \kappa\ \zeta\ \theta\rangle, \sigma, \Theta, \mathrm{T}, \mathrm{M}\rangle, \\
&\qquad \underline{\texttt{let}}\ \omega = (\kappa\ \textit{unspecified}\ \sigma\ \Theta)\ \underline{\texttt{in}}\ (\mathcal{S}[\![\omega\lfloor_1]\!]\ \omega\lfloor_2\ \omega\lfloor_3\ \omega\lfloor_4 \S\mathrm{T}\ \omega\lfloor_5 \S\mathrm{M})
\end{aligned}$$

The *suspensive?* function determines whether a context ζ should be suspended in the presence of signals σ and stores Θ. If the context is empty, it is not suspensive, otherwise we look at the first clause ι: if it is a `when` clause with a non-emitted signal the context is suspensive, it is is a `ref` clause with a store not present in Θ the context is suspensive, otherwise we look if the rest of the context ζ_1 is suspensive.

$$\begin{aligned}&\textit{suspensive?} = \lambda\zeta\sigma\Theta.\zeta{=}\emptyset \rightarrow \text{false}, \underline{\texttt{let}}\ \iota = \zeta{\downarrow_1}, \zeta_1 = \zeta{\dagger}1\ \underline{\texttt{in}}\\ &\quad(\textit{when?}\ \iota) \rightarrow (\textit{when-signal}\ \iota){\notin}\sigma{\vee}(\textit{suspensive?}\ \zeta_1\ \sigma\ \Theta),\\ &\quad(\textit{ref?}\ \iota) \rightarrow (\textit{ref-store}\ \iota){\notin}\Theta{\vee}(\textit{suspensive?}\ \zeta_1\ \sigma\ \Theta),\\ &\quad(\textit{suspensive?}\ \zeta_1\ \sigma\ \Theta)\end{aligned}$$

The scheduler's entry point is *schedule-instant*, which takes a list of threads to schedule and an initial set of signals (those emitted during the instants) and schedules the current instant. It starts by building the set of stores present for this instant into Θ, then invokes the intra-instant scheduler function *s-micro-instant*. This function returns in ψ a list of signals σ_1 to emit at the next instant, and a list of thread T_1 to schedule at the next instant. If there are no more threads to schedule the scheduler terminates by calling *exit*, otherwise it recurses to schedule the next instant.

$$\begin{aligned}&\textit{schedule-instant} = \lambda\mathrm{T}\sigma.\underline{\texttt{let*}}\ \Theta = (\textit{map}\ \textit{thread-store}\ \mathrm{T}), \psi = (\textit{s-micro-instant}\ \mathrm{T}\ \sigma\ \Theta\ \emptyset),\\ &\quad\sigma_1 = \psi{\lfloor_1}, \mathrm{T}_1 = \psi{\lfloor_2}\ \underline{\texttt{in}}\ \mathrm{T}_1{=}\emptyset \rightarrow (\textit{exit}), (\textit{schedule-instant}\ \mathrm{T}_1\ \sigma_1)\end{aligned}$$

The intra instant scheduling function iterates over each thread in T to schedule it with $\mathcal{S}$ until the instant is finished. The instant is finished when *instant-finished?* returns true. If the instant is done, we return the result of *end-of-instant*, which executes the actions to do at the end of instant. If the instant is not finished we schedule each thread in T using *s-threads*, which returns a list consisting of the threads' new statuses, the new signal environment, present stores, set of threads to create and migrations to trigger. Using these results we recurse and reschedule all threads (together with the newly created threads) until we reach the end of instant.

$$\begin{aligned}&\textit{s-micro-instant} = \lambda\mathrm{T}\sigma\Theta\mathrm{M}.(\textit{instant-finished?}\ \mathrm{T}\ \sigma\ \Theta) \rightarrow (\textit{end-of-instant}\ \mathrm{T}\ \sigma\ \mathrm{M}),\\ &\quad\underline{\texttt{let}}\ \Omega = (\textit{s-threads}\ \mathrm{T}\ \sigma\ \Theta\ \emptyset\ \mathrm{M}\ \emptyset)\ \underline{\texttt{in}}\ (\textit{s-micro-instant}\ \Omega{\lfloor_1}\S\Omega{\lfloor_4}\ \Omega{\lfloor_2}\ \Omega{\lfloor_3}\ \Omega{\lfloor_5})\end{aligned}$$

The *s-threads* function simply schedules each thread in T and builds up the sets $\mathrm{T_n}$ of new threads to create, and M of migrations to trigger. It returns a list with the results of the threads T scheduling, the new sets of signals σ and stores Θ, the threads created during this round of scheduling $\mathrm{T_n}$ and migrations M.

$$\begin{aligned}&\textit{s-threads} = \lambda\mathrm{T}\sigma\Theta\mathrm{T_n}\mathrm{M}\mathrm{T_s}.\mathrm{T}{=}\emptyset \rightarrow \langle(\textit{reverse}\ \mathrm{T_s}), \sigma, \Theta, \mathrm{T_n}, \mathrm{M}\rangle,\\ &\quad\underline{\texttt{let}}\ \Omega = (\mathcal{S}[\![\mathrm{T}{\lfloor_1}]\!]\ \sigma\ \Theta\ \mathrm{T_n}\ \mathrm{M})\ \underline{\texttt{in}}\ (\textit{s-threads}\ \mathrm{T}{\dagger}1\ \Omega{\lfloor_2}\ \Omega{\lfloor_3}\ \Omega{\lfloor_4}\ \Omega{\lfloor_5}\ (\textit{cons}\ \Omega{\lfloor_1}\ \mathrm{T_s}))\end{aligned}$$

We deem an instant to be finished when each thread marked as `ready` is suspensive. The other thread statuses `terminated` and `pause` are indeed not scheduled anymore.

$$\textit{instant-finished?} = \lambda\mathrm{T}\sigma\Theta.\forall\tau{\in}\mathrm{T}, (\textit{thread-ready?}\ \tau) \rightarrow$$

$(\mathit{suspensive?}\ (\mathit{thread\text{-}context}\ \tau)\ \sigma\ \Theta)$, true

During the end of instant we have many things to do. First we filter out the terminated threads and change the statuses of `pause` to `ready` by calling *next-ready* on the threads T. We then map those threads by preempting those that need to be, which gives us the list of threads T_1, ready for the next instant. We then invoke *emigrate* with these threads and the migration orders M. This function will send the threads marked for migration and return a list ψ of two values: the set of signals emitted during migration to notify success or failure, and the list of threads which either did not migrate or failed to migrate and had to be respawned. We then pass those two values to *immigrate* which incorporates immigrant threads and add to the set of signals those that notify their successful migrations.

$\mathit{end\text{-}of\text{-}instant} = \lambda T\sigma\Theta M.\underline{\texttt{let*}}\ T_1 = (\mathit{preempt}\ (\mathit{next\text{-}ready}\ T)\ \sigma\ \Theta),$
$\psi = (\mathit{emigrate}\ M\ \emptyset\ T_1)\ \underline{\texttt{in}}\ (\mathit{immigrate}\ \psi\lfloor_1\ \psi\lfloor_2)$

The *preempt* function simply maps *preempt-t* on each thread in T.

$\mathit{preempt} = \lambda T\sigma\Theta.(\mathit{map}\ \lambda\tau.(\mathit{preempt\text{-}t}\ \tau\ \sigma\ \Theta)\ T)$

In order to preempt a thread, we have to look at its context ζ, and more precisely, the context ζ_{watch} of its outermost satisfied preemption clause (a satisfied preemption clause is one for which the preemption signal was emitted). If no such context exists, the thread τ is returned unchanged since it does not need to be preempted. Otherwise we extract the satisfied preemption clause in ι and the list of protection clauses ζ_{protect} which need to be called between the thread's current context and ι. We then change the thread's continuation to unwind all these protectors until it calls the preemption clauses' continuation.

$\mathit{preempt\text{-}t} = \lambda\tau\sigma\Theta.\underline{\texttt{let*}}\ \zeta = (\mathit{thread\text{-}context}\ \tau),\ \zeta_{\text{watch}} = (\mathit{last\text{-}preempt}\ \zeta\ \sigma\ \Theta\ \text{false})\ \underline{\texttt{in}}$
$\neg\zeta_{\text{watch}} \rightarrow \tau,\ \underline{\texttt{let*}}\ \iota = \zeta_{\text{watch}}\downarrow_1,\ \zeta_{\text{protect}} = (\mathit{filter\ protect?}\ \zeta\backslash\zeta_{\text{watch}})\ \underline{\texttt{in}}$
$\langle\texttt{ready}\ \lambda\varepsilon\sigma_1\Theta_1.(\mathit{unwind\text{-}to}\ \zeta_{\text{protect}}\ (\mathit{watch\text{-}kont}\ \iota)\ \sigma_1\ \Theta_1)\rangle$

Determining the context of the outermost satisfied preemption is simple. If the context ζ is empty, we return r, which holds the last satisfied preemption or false. Otherwise we extract the inmost clause ι and check whether it is a satisfied preemption clause. In order to match, it needs to be a preemption clause, whose signal has been emitted, and whose context ζ_1 is not suspensive (suspended preemption clauses are not preempted). If it matches we recurse to ζ_1 and keep the clause. It it does not match we recurse but keep the last matching clause.

$\mathit{last\text{-}preempt} = \lambda\zeta\sigma\Theta r.\zeta{=}\emptyset \rightarrow r,\ \underline{\texttt{let}}\ \iota = \zeta\downarrow_1,\ \zeta_1 = \zeta\dagger 1\ \underline{\texttt{in}}$
$(\mathit{watch?}\ \iota)\wedge(\mathit{watch\text{-}signal}\ \iota){\in}\sigma\wedge\neg(\mathit{suspensive?}\ \zeta_1\ \sigma\ \Theta) \rightarrow$
$(\mathit{last\text{-}preempt}\ \zeta_1\ \sigma\ \Theta\ \zeta),\ (\mathit{last\text{-}preempt}\ \zeta_1\ \sigma\ \Theta\ r)$

The *next-ready* function takes a list of threads T, filters out the terminated threads and changes the status of `pasue` into `ready` so that threads waiting for the next instant become eligible again for scheduling.

next-ready = λT.(*filter-map* $\lambda\tau$.(*thread-terminated?* τ) $\rightarrow$ false,
$\neg$(*thread-pause?* τ) $\rightarrow \tau$,
$\langle$`ready` (*thread-kont* τ) (*thread-context* τ) (*thread-store* τ)$\rangle$ T)

The emigration function *emigrate* takes a list of migration orders M, a set of signals to emit at the next instant σ and the list of threads currently present T. If there are no migration orders in M we return the set of signals σ and the list of threads T. Otherwise we extract the first migration order into μ, make a list of threads in T_{group} which share the store we are migrating θ, and attempt to send the group using the *send* function, which returns true if the migration was successful, false otherwise. If the migration was successful, we recurse with the rest of the migration orders, adding the migration signal marked as successful to σ, and removing T_{group} from the list of threads T. If the migration was not successful we recurse with the rest of the migration orders, adding the migration signal marked as failed to σ, and removing T_{group} from the list of threads T only if we did not want to respawn the failed migrators, otherwise we leave them in T so they can respawn.

emigrate = λMσT.M=$\emptyset \rightarrow \langle\sigma$, T$\rangle$, `let*` μ = M$\lfloor_1$, dest = $\mu\lfloor_2$, $\theta = \mu\lfloor_3$,
timeout = $\mu\lfloor_4$, respawn = $\mu\lfloor_5$, sig = $\mu\lfloor_6$, T_{group} = (*group* θ T) `in`
(*send* dest T_{group} timeout sig) $\rightarrow$ (*emigrate* M$\dagger$1 σ§(*success* sig) T\\T_{group}),
(*emigrate* M$\dagger$1 σ§(*failure* sig) respawn $\rightarrow$ T, T\\T_{group})

The immigration function *immigrate* takes a set of signals σ to emit at the next instant and a list of threads T to schedule at the next instant. If there are no incoming agents available it returns σ and T. Otherwise we receive incoming agents and their corresponding migration signals into ψ. We then recurse by appending their migration signals marked as successful into σ and the new threads to T. The actual implementation might block waiting for new agents if T is empty, since there would be no thread to schedule and instead of exiting the program it might be desirable to wait for new agents.

immigrate = $\lambda\sigma$T.(*no-incoming?*) $\rightarrow \langle\sigma$, T$\rangle$, `let` ψ = (*receive*) `in`
(*immigrate* σ§(*success* $\psi\lfloor_1$) T§$\psi\lfloor_2$)

3.4 Implications of Migration

We now know everything about the language's primitives and scheduling, but we still have the gray area which is migration. The fact that a thread can leave one site for another involves a process of serialisation on the departure site and integration on the target site which is not straightforward to integrate in Scheme.
In this section we will be talking about how migrating threads part from their site, and how they are integrated on the new site.

3.4.1 The parting of ways

When agents migrate from one site we expect them to arrive on the new site and keep working. This is obvious, yet the question of how this is done involves the question: what

does an agent need to leave with in order to keep running when it arrives?
The semantics described in the previous section offers some hints about that: the evaluation of an expression needs an environment, a context, a store, a set of signals and a set of present stores. Obviously the sets of signals and present stores are global and belong to a site, so they should not be migrated with the agents. In our semantics these globals are never captured, and always obtained from the scheduler, so there is no need to worry about them: the agents will get the local sets of signals and present stores from the target site's scheduler.
The context and the store are obviously something that belong to the agent which is migrating in whole, and should thus be packed along with the agent.
The environment on the other hand is not obvious: it is inherited by new threads, yet just like closures it is shared with the thread creator. Although the semantics does not discriminate between the global variables (introduced in Scheme with toplevel `define`) and local variables (the lambda arguments) we should make a distinction as this point in the discussion.

Local variables

Local variables are those introduced by lambda arguments and `let` bindings. They are those variables which are captured by functions and threads. Contrary to ML they are not read-only and can be modified. Moreover modification of a local variable should be visible to all those which captured its environment.
Since Boudol's ULM paper describes its integration in an ML language with no global variables, no module system and read-only local variables, it is assumed that local variables are copied to the target site, and references describe the only read/write variables.
This is not the case in Scheme however, since local variables are not read-only. Making every local variable a reference is not a practical solution for Scheme since it would mean that after a migration, an agent's captured local variables would all become blocking, even for read-only access. We found this solution counter-intuitive, so we did not adopt it.
Some compile-time analysis could have marked the local variables which are in practice read-only, and those which are modified. With this analysis we could have made all modifiable variables into references a bit like in ML, but unlike in ML, creation and access to those references would not be explicit, and differentiating visually which variable in the code is read-only or a reference (which really means: which can block and which cannot) would be hard. For all those reasons we chose not to do it this way.
The only problem with local variables and migration is the *unification* phase: what happens when two agents capture the same variable and modify it after being parted by migration? Then if these two agents rejoin the same site, what should happen with both instances of the captured variable?
Since we want reference access to be explicit, and they already answer the unification problem (with their blocking semantics), we chose to not make local variables potentially blocking, and not worry about them being unifiable. Our answer then is to copy (and thus duplicate) the captured environments when migrating, and never unify them even if they get the chance.
This means that if a thread A and an agent B share a captured local variable ν, after B migrates it will live with a copy ν_1 which will still be modifiable by B, but even if both threads come back to the same site, ν and ν_1 will never be unified, and neither thread will see the modifications made by the other. In practice this means that after migration

a thread will be able to take as much as needed about the environment to be able to keep running on the new site, which is what is intuitive.

Global variables

Global variables are different from local variables in two ways relevant to us: their values are not captured, but can be obtained from the Scheme system by name (like a dictionary), and they can be grouped by modules. Since Boudol's ULM paper describes neither global variables nor modules, we had to figure out what to do with them with regards to migration.
The fact that they can be obtained by name and are not captured means that after migration it is possible to do *dynamic rebinding*: that is to say that if an agent was using a global variable ν_{A} on site A, after it migrates to the site B it might want to use the global variable ν_{B} in its stead.
The fact is that while it is possible to do dynamic rebinding of global variables, it may not be what is wanted in all cases, for example when the global variable has no counterpart on the destination site. To that end we decided to make it explicit which global variables are supposed to be *ubiquitous*: have a counterpart present on all sites, and thus use dynamical rebinding. Since global variables belong to modules in our implementation, and we can expect generally modules to be a whole with global variables depending on one another, we chose to make the distinction of ubiquity on the module level: a module is either ubiquitous or not.
We thus redefine the module declaration as such:

```
<module-definition> ==> (module <module-name>
                          <module-spec> ...)
                        | (ubiq-module <module-name>
                          <module-spec> ...)
```

Modules declared with the `module` header are not ubiquitous, and any agent which uses such a module's global variable will migrate with a copy of that variable, exactly in the same way as local variables are cloned on migration: there will be no unification later on. On the other hand, global variables belonging to modules declared with the `ubiq-module` header will be dynamically rebound to the module's counterpart on the destination site. All of ULM's runtime library is defined in ubiquitous modules.

Ubiquitous values

Just like variables, some values behave differently during migration. Numbers, booleans symbols for example, since they are unmodifiable values, can be seen as ubiquitous values: upon arrival they are not deserialised as new values but are replaced by their local counterparts. Pairs, vectors and strings on the other hand are not and they are cloned when captured as part of the migration process.
ULM-specific values also have special migration semantics. Signals are ubiquitous values because they are used as synchronisation mechanisms that must keep on working after migration so that an agent can communicate with the site it visits. References are more peculiar: since they are local only when the reference's store is local, they can change status when stores leave or join the site. They are also ubiquitous: two references to the same part of a store are unified during immigration. Note that it is an interesting kind of ubiquity

since all references to the same part of a store represent the same reference on all sites, even though it is local only to one site at a time (that which has its store).

3.4.2 Things that need more work

Like every language, ULM is not perfect (yet), and we have noticed some limitations we would like to address in a future version of ULM.

Custom serialisers

Right now ULM knows how to serialise all the data types you can create in ULM, but for some user-created data types (such as mixins or native/extern data) it would be preferable to register custom serialisers and deserialisers which could be responsible to that data type's serialisation. This is a very popular feature of Java even though the language itself does not support migration directly.
The practical aspects of such serialisation are complex, and not to be addressed lightly. Because migration is done at the end of instant, and we are migrating the agent and its threads, these threads cannot be the ones doing the serialisation. Similarly on the arrival site the arriving threads cannot be used for deserialisation. So the serialisation and deserialisation of special user structures (presumably mixins since that is the only user-definable data structure) must be done in a special *serialiser* auxiliary thread.
The system could then invoke a registered `serialiser/deserialiser` function in this thread. The serialiser function would take data structure and return a string or a vector of bytes for the serialised data. The deserialiser would take a string or a vector of bytes and return the deserialised object. An alternative would be to associate mixins to these functions by marking mixins in need of special serialisation using inheritance as in Java: mixins which inherit from a special `serialisable` mixin would be treated specially by the system, which would then invoke the mixin's `serialise` and `deserialise` methods. These methods would serialise the current mixin in the case of serialisation, and for deserialisation the mixin would be allocated by the system, and the deserialising method would fill up its fields.
Should the serialisation or deserialisation fail from the *serialiser* thread, it would cause the agent's migration to fail and it would be handled as described earlier for network failures. Since any ULM code can cooperate or wait on signals, doing so in the *serialiser* thread would delay the migration, so it may make sense to create a *serialiser* thread per agent migration, so that one agent's serialising code could not delay other agents' migration.
Of course, the system would have to provide functions which serialise and deserialise ULM primitive data types, such as closures and environments, which may require a significant modification of our serialisation process described in the next chapter.
We have not yet taken enough time to study this feature, but we have the feeling it is feasible and it would be a nice addition if we can integrate it cleanly.

Custom ubiquitous values

Right now the system already knows which values are ubiquitous, but in some cases this is not enough. One could create a constant value which should be ubiquitous. For example, the base mixin hierarchy: we already described that each mixin type is itself a mixin value

describing the mixin type. This makes for a lot of values which are common to all sites which share the ubiquitous module of the base mixin types.
One solution we're thinking of is to register such values to the system, where they would be attributed a universal unique identifier (UUID) which would be used for unification. Whether the user has to mark these ubiquitous values itself in the program, or whether the compiler or runtime can automatically assign them in such a way that all sites would assign them similarly, is unknown at this point. How one would annotate such values in the code is also unknown.

Ubiquity failures

When we have successfully registered all ubiquitous modules and values, what should become of an agent arriving on a site where the ubiquity is not preserved? If a ubiquitous value or module is not present on the target site, because the programmer has failed to make his ubiquitous modules available on each site, or if these ubiquitous modules are out-of-date and don't have the same interface. In such cases, what should happen?
A possibility is to declare the migration a failure, and notify the origin site of that failure (if it is a safe migration, possibly respawning the agent back there). Another possibility would be to make the agent wait for the arrival of the missing value or module. Since the agent might be undeserialisable in such a case it might be possible to designate handlers for such cases, which could be migrated (by safer means?) or be pre-installed on the site. Fixing the problem of a missing module would however require a way for agents to migrate modules, or to load them from means different from the site's, like Java's ClassLoader objects.
This is again an area which poses many questions and deserves some answers.

Chapter 4

Implementation: Scheme

Les détails font la perfection, et la perfection n'est pas un détail.

– Leonardo da Vinci

It is a mistake to think you can solve any major problems just with potatoes.

– Douglas Adams

In this chapter we leave the clear open skies of design and specifications for the misty caves of implementation. As beautiful as the *big picture* can be, the inner workings of an implementation can be equally beautiful in their own way, and definitely as interesting for some.
In order to ease the process of delving into the implementation we have decided to separate the presentation in two parts: the first relates to *Scheme* generally and is presented in this chapter, while the second concentrates specifically on ULM and is presented in the next chapter.
In this chapter we will start by describing the virtual machines that allow ULM programs to run, followed by the bytecode used by these virtual machines and finally talk about the bytecode is formatted after compilation.

4.1 Two Virtual Machines

ULM programs are compiled by a compiler written in Bigloo Scheme, into a bytecode format we will describe in 4.2. This bytecode is then interpreted by a ULM virtual machine. In this section we will explain why we are using not one but several virtual machines, and what their differences are.

4.1.1 Why a virtual machine?

A language such as ULM, which supports thread creation, requires a specific implementation that offers such threads. There are several ways to do that[1]. We have however a very strong requirement: our threads have to be able to migrate from one computer to another, possibly with very different architectures.

[1] Isn't there always?

Should we compile ULM natively?

Compiling ULM to native code would make it hard on different levels. Firstly, the compiled code itself would be difficult to migrate, since native code is known to be very *unportable* since it is written specifically for an architecture's processor instructions and operating system interfaces. Secondly, native implementations of thread often use data structures (usually the stack) which are not reified, and hard to examine and copy between computers. There are ways around these problems of course, like recompiling the agent's source code at every site we migrate to, or annotating the runtime structures such that they would be reifiable and copyable between machines, but this represents a huge effort to write such systems for every possible architecture (even for only the most used architectures).
Compiling code to existing virtual machine bytecode such as Java or .NET would be easier, since the bytecode is portable. On the other hand there is no stack reification on such platforms, and even though there are ways to work around this [PCM+05] [AAB+05], the fact that we want to be able to handle large numbers of ULM threads and agents means that we have to give up on using those platforms' threads, and therefore stacks. There is little value in compiling ULM to Java or .NET bytecodes if we are not going to use the virtual machine's support for threads and stacks as we would have to implement all the stack and memory handling ourselves.

Should we use the interpreter's continuation?

In order to both be portable and concentrate on higher-level problems than architecture-specific discrepancies, we have decided not to compile ULM to native code, but write a ULM interpreter in a portable language. Even then, the requirement of being able to create and migrate threads proved to be determinant.
Traditional academic Scheme interpreters are usually written in Scheme themselves, and their simplest form consists in having an `eval` function, which evaluates an expression by using recursion. For the sake of clarity, we will call the Scheme used to write the interpreter the *Host Scheme*. What this means is that when we are evaluating the function `f`, which invokes the function `g`, we are evaluating the body of `g` while evaluating the body of `f` by recursion. If this recursion is not terminal[2] the Scheme Host has to store information about what is left to do in `f` after `g` returns.
Now depending on the Scheme Host this may be located in stack frames, or in continuation functions. But while the idea would sound delightful (others would say weird), the Scheme Host itself cannot be interpreted by another Scheme Host indefinitely. There has to be a layer at the bottom of the interpreters, which is native code. Let us reduce the number of layers to zero, and assume the Scheme Host is implemented in native code (of course it can be written in Scheme and compiled into native code). If we have to look at the Scheme Host's stack it will be the native stack and we fall into the same problem we described earlier. If the Scheme Host is using continuations, they are most likely also compiled into native code and are thus equally hard to inspect. We do not want to rely on the Scheme Host's implementation to store data we need for migration, so we can rule out recursive interpreters.
An other typical scheme interpreter in scheme will be written in *Continuation Passing Style* (CPS), where the stack is not used for recursion, but the continuation of the interpreter is

[2] A terminal recursion is a recursion in which it is not necessary to return to the caller function after returning from the callee.

built explicitly by constructing closures representing the continuations. This is, however, still not a good solution for us since migrating a thread would mean migrating these Scheme Host closures, which we already deemed to be implemented natively, and thus not portably. What we need in order to support threads and migration in a ULM interpreter is for this interpreter to handle the ULM continuation (to be able to save and restore it portably) without resorting to the Scheme Host's continuation.

Should we abstract our own continuations?

After having ruled out relying on the Scheme Host (or actually any Host language) to hold continuations or stacks for ULM, we are left with the option to implement our own stack-handling, and code-evaluating interpreter. There are several ways to do this, but in order to abstract the underlying host language or architecture, to interpret a language with full control on its evaluation and continuation, it is common to use a *virtual machine* (VM).
Virtual Machines consist in a loop iterating over *bytecode* to execute instructions which do not require the Host's stack. The bytecode is compiled from the original source code (ULM) and is generally very portable. Since the virtual machine provides its own stack, registers, we have a solution for migration: the interpreter knows how to create, inspect and install all the data structures needed for migration without relying on the Host language.
The only thing left to do is find a Host language which is portable, in the sense that the Virtual Machine will be able to run on as many architectures as possible.

4.1.2 The first virtual machine

While we could have adapted one of many Scheme VM for ULM, we felt it would make more sense to start from (almost) scratch with support for threads and mobility at the very core of our interpreter, then add features on top of this sound core. Indeed, such a fundamental change as adding threads and mobility to a Scheme VM with a large feature-set may be tedious and break the VM's coherence in ways hard to foresee.
We started out with a simple academic compiler and virtual machine from Queinnec's L.I.S.P. [Que96], which supports standard Scheme primitives in a clear and straightforward way. We chose as Scheme Host the Bigloo language so that our virtual machine and compiler could be either compiled natively for best performance, or to Java or .NET for good portability and fair performance.
Our work on the compiler has been to integrate ULM primitives, modules, integrate our bytecode file format, support macros, add debugging information, (some) type-checking and nicer error reporting. Some of these features are described in details in this chapter and the following one.
Our work on the virtual machine has been a mirror of the work on the compiler (all the compiler's new features have to be supported by the VM), as well as supporting threads, serialisation and migration, and adding many Scheme primitives to the runtime. This, and several other features of what we call the Bigloo VM will be described in this chapter and the following one.

4.1.3 Why two (3?) virtual machines?

At one point in the project we began wondering where ULM agents could go in places other mobile agents have not gone before. FairThreads-type of cooperative threads have

always been touted as embeddable, because of the lightweight implementation and memory footprint they require. ULM being based on such a model, we always had in mind that ULM agents were light too, and could be useful on small embedded devices.

Targeting an embedded platform

In order to put our theory to the test, we set out to find an embedded platform we could target. To this day, the most widespread portable embedded platform is probably Java 2 Micro Edition (J2ME), if simply for the fact that nearly all mobile phones on the planet support it. The facts that in our offices we had several such phones with good support for the platform, that an almost full-fledged Java runs on it, and that it is very open and documented made it the platform of choice for our experiment.
We speculate that another very popular embedded platform (although less portable somehow) is Linux on ARM, but we did not choose it for lack of the presence of such a platform in our office (even though everyone has at least one at home in one way or another), made it a more difficult target. At the end of our thesis, many newer ultra-portable computers (PDA) are based on ARM and Linux, and newer phones are also expected to go in this direction, so in the near future we can expect this platform to outgrow the number of J2ME platforms. We are confident though that our Bigloo VM will be easy to port to such devices with minimum changes (or none).

The J2ME platform

Since our first VM was written in Bigloo, and can be compiled to the Java backend, our first attempt was to use the same VM for the J2ME platform. But while J2ME supports the complete Java language, it does not support the complete runtime API. Far from it: from the base API only the three packages `java.io` (with no filesystem support), `java.lang` (without ClassLoader) and `java.util` (prior to collections) are present. For those not expert in the Java API, this means no network I/O, no reflection, no dynamic class loading, and no collections.

Adding a J2ME backend to Bigloo

Historically the Java API has grown over the years, by adding features at every major revision, and many projects which use Java and accept any version of the Java API use various tricks to be able to work on older versions while benefiting from the newer versions' improvements. Bigloo has such a capability: at runtime it is able to use codepaths which vary depending on the Java API available. This is only possible in Java because classes are loaded Just-In-Time, and method and variable references are also checked Just-In-Time. This means that for an older version of Java, as long as we don't execute the codepath using the newer classes, methods or variables, there will be no problem.
Initially we set out to providing codepaths for J2ME, thereby avoiding the parts of the Java API not available on that platform. This is more difficult than adding codepaths for new features since we're removing features that have been there since the first version of Bigloo's Java backend. But this turned out to be a non-possibility, since J2ME requires the JARs to be *verified* in advance, which means that every codepath in the JAR will be checked, and obviously the codepaths which use non-J2ME features would not be verifiable.

Pruning Bigloo's runtime for J2ME

There is a simple solution to that problem: prune the offending codepaths from the JAR at compile-time, thus making a special Bigloo runtime which could not work on anything else than J2ME. At this point we noticed another problem: the device we were testing on supports only JARs of up to 128 kilo-byte size. But Bigloo's runtime JAR is 1 mega-byte, and that does not account for the size of our ULM Virtual Machine JAR.
In order to make this fit, we also needed to trim the Bigloo runtime so that it would both fit and leave enough space for our VM. Fortunately, there are entire parts of the Bigloo runtime which make no sense on J2ME: files for instance have no equivalent in J2ME, since there is no filesystem accessible from J2ME. Then there are many subsystems of the Bigloo runtime which are not needed by our ULM VM: the evaluator, the parsers, many I/O libraries, and within many subsystems we are not using every function.
It turns out that removing subsystems is not easy, as many are inter-dependant. We spent several weeks attempting to prune the runtime of subsystems we did not need, but the size gain was not enough. Then we turned to removing individual functions we did not need. To that end we started looking at various JAR optimisers which offer the ability to traverse all codepaths and prune unused methods and functions from a JAR. But ironically, work undergone in Bigloo to reduce the runtime size in Java prevented us from using such a tool. In Bigloo's Java backend, modules are represented by Java classes, and global variables by class fields. A straightforward implementation would represent Closures with a `closure` Java class, with an `invoke` method which in turn invokes the proper Java method in which the closure's code would have been compiled. In short, every different closure in Bigloo would be represented by a subclass of `closure` with a specific `invoke` method. But this type of implementation was not used by Bigloo because it needs to generate the same number of class files as there are closures in the Bigloo runtime, and having too many small class files is bad for size. One of the reasons for this is that class files each have a constant pool [LY99] and having less and larger class files removes many duplicate constants from the pool, thus diminishing the total size of the JAR [BHV98] [Pug99] [RMH99].
In order to save space and reduce the number of class files produces by its Java backend, Bigloo stores several closures in a single Java method by using a switch, and represents closures with instances of a single `closure` class with an index pointing at the switch entry representing its code [SS02]. This is very efficient in reducing the number of classes produced, and thus the size of the runtime JAR, but it has the disadvantage that since every closure has its code located in a single method (more precisely one method per closure arity) all the programs we tested to remove unused code failed to remove the unused code in these switches, even if they managed to remove the only closure instances that were pointing at the switch entry.
At this point we felt that it would be difficult to attain the 128k limit on the code size with the Bigloo runtime, especially when we had to add our VM to the equation, since both were not written with such a small footprint in mind.

4.1.4 The Java VM

Since it was too hard to get our first virtual machine to work with the J2ME constraints, we set out to write a new virtual machine directly targeted at J2ME. The process of writing this new virtual machine, which has to execute the ULM bytecode with exactly the same semantics as what we shall from now on call the Bigloo VM, involved porting some parts

of the VM, while rewriting some other parts and sometimes even completely rethink some components.
Since it is hard to quantify the RAM use of our future VM on the J2ME device, we first set out to not break the 128k limit on code size. To this end we attempted to reduce the number of classes required for our VM to the bare minimum. In some cases this can be easy: when using classes as modules for different units of logic, architecture can be pushed back and we can merge several units of logic to a single class, even if we have to prefix some methods with the unit name to help make a distinction.

Merging classes

For non hierarchical structures of data (such as classes with no parent class and no subclass), we can also group them in the same class with a field used to differentiate between the types of data they represent. For example, a type A with an integer field, and a type B with a string field can be represented by a class C with an integer, string and boolean fields, the boolean field being used to specify whether the instance of C is used as a type A or B. This has the inconvenient of using extra RAM for the unused fields (since there is no *union* specifier in Java like there is in C), and requires an extra check at runtime (accessing the boolean field) to know the exact type of the object, but it does reduce the number of classes. We used this trick in several places for internal data types, and for representing symbols and keywords.

Flattening hierachies

This trick can also be used for some hierarchical structures of data by flattening the types. For example, we have a type for threads (which have a parent), a type for agents (which have no parent, but have children) and both types share many common fields such as the stack. Architecturally, we should have an abstract super class VThread with common fields such as the stack, with two subclasses Thread (with a parent field) and Agent (with a children field). If we flatten everything, we get a VThread class with the common fields, a parent and children fields, and a new boolean field telling us whether the VThread is a Thread or an Agent. We can even save this boolean field by noting that Agent threads have their parent set to `null` while non-agent Threads must have a non-null parent.

Representing primitives

Of course, in order to represent the virtual machine primitives, which are similar to how Bigloo compiles closures in that an instance of each primitive has to map to some code, we used the same trick as Bigloo. Each primitive is an instance of the `Primitive` class with an index pointing to a switch where all the primitives' code is written in the same method (actually as in Bigloo a method by primitive arity). We will look at this in more detail as the same framework is used for native modules which we describe in chapter 6.

VM Size success

At the end of our implementation, we have a VM JAR of 96k of Java code (once zipped) for a total of 26 classes. This includes some extra classes required in order to be able to extend the VM to run on a regular Java platform (J2SE). This leaves us 32k for ULM libraries.

Since the runtime takes 11k (in the JAR) in our experience it leaves enough space for agent applications.

Different runtime strategies

The J2ME phone we used for development (a Nokia 6230) is listed as having 1m of heap size. Because J2ME does not include any API to query the available or used memory, because J2ME devices can be expected to have different Java implementations, and because they are hard to instrument from a computer in order to determine empirically the memory footprint of our ULM VM, it is hard to speculate about our memory footprint. One thing is sure though: we should strive to have a low memory usage because 1m is not much (especially compared with modern personal computers).
We know that we have some overhead due to the various tricks we used to limit the number of classes in the VM. In order to compensate this overhead as well as prevent overall memory usage, we changed some algorithms when writing the J2ME VM. For instance the scheduler is not exactly the same as the Bigloo VM scheduler, which uses many caching techniques and distributes the End Of Instant scheduling within the instant whenever possible. The scheduler implementation in the J2ME VM is simpler, and closer to the semantics. The module in charge of sending and receiving agents is also very different in order to reduce the number of threads running at the same time.

Road bumps with J2ME

After all the effort undertaken to make the J2ME VM, we had many disappointments with the J2ME emulators and the poor state of J2ME implementations on our mobile device. First of all J2ME devices have two ways to communicate on the network: TCP/IP and bluetooth. TCP/IP is done through the GPRS link to the phone service provider (and every connection has to be paid for). Bluetooth on the other hand is akin to a serial wireless interface, and can be used to communicate for free to an other phone or a computer with bluetooth support. Of course we used bluetooth as our preferred agent migration link, since we did not intend on paying for every migration.
Under linux though, the J2ME emulator provided by Sun provides only a virtual bluetooth link, which does not work with the real bluetooth interface, so it was impossible to test migration of agents using bluetooth on the emulator. Furthermore, testing on the phone itself is hard because there is no remote debugger, and not even a console output to view error messages. When something wrong happens, we see a stack trace (if we are lucky) on the phone, with no line information. This is very frustrating and very time-consuming.
Our only means to know what is going on on the phone is to open a constant bluetooth serial link between the phone and the computer in which we print debugging messages while the VM is being executed. While that provided us with the minimum debugging tool which is an output console, it bothers the bluetooth stack of the phone and the part which negotiates the bluetooth channel (similar to the port number in TCP) stops working when using such a debugging link. We know that the phone's manufacturer is aware of the issue, but there is no fix available, and we had to resort to various unportable tricks (in the sense that other J2ME devices would not work anymore) to get bluetooth migration working while debugging.
To further make the J2ME development interesting, we found that the phone in question froze and rebooted every so often, even when executing exactly the same ULM programs

with no migration involves, in different invocations of the J2ME JVM.

The third VM, the J2SE VM

In order to ease the development process (sometimes to make it possible at all), we developed a subclass of the J2ME ULM VM targeted at J2SE (the standard Java). This required the use of subclasses for some components such as migration, because TCP/IP network and bluetooth serial accesses in J2SE is radically different than in J2ME. J2SE also provided us the means to add several Scheme primitives such as filesystem operations, and the ability for our VM to load ULM modules directly from the filesystem, instead of having to pack them in the same JAR as the VM on J2ME.
With this J2SE VM we were able to test and debug our VM on the local computer, which made development much easier. Further along the project we realised that this J2SE VM was useful for other types of embedded devices: those with linux or Windows Mobile Edition which support a full J2SE while having a limited disk space (although much less constrained with 32m and up). Although our Bigloo implementation would work on such platforms, the J2SE VM still takes a lot less disk space.

4.2 Bytecode compilation and interpretation

Now that we have explained why we need not one but several virtual machines, we will describe how we compile ULM to bytecodes and how they are interpreted. ULM's bytecodes as well as its compilation and interpretation are based on Queinnec's bytecode in L.I.S.P. [Que96]. Rather than describe formally or informally what is a boring list of virtual machine structures and bytecode specifications, we have chosen to present, like Queinnec, the 109 bytecodes and the virtual machine step by step, by showing how we compile each Scheme[3] primitive one after the other. Hopefully introducing each concept one at a time will make the presentation less dense.

4.2.1 Some required introduction

In order to start introducing elements of compilation and execution for the ULM bytecode, there are a few terms and structures we have to outline first. We have chosen to list here several structures used in the VM, without explaining them in details yet, so that when later we talk about changing, say, the *current escaper*, the reader will be able to look up that the *current escaper* is a member of the *thread* structure.

Modules

As we have previously described, ULM code is organised in modules. Each module contains a header that gives information about the module (name, imports, exports, ...) and a toplevel of Scheme expressions to be evaluated when the module is loaded. From a compilation and interpretation point of view, ULM modules are composed of:

- a name,
- an ubiquity attribute,

[3] And in the following chapter ULM primitives.

- a list of imported modules,
- a list of exported symbols,
- a list of global variables (%GLOBALS),
- a list of constants (%CONSTANTS),
- a list of bytecodes,
- a list of function descriptors.

Once again, every one of these members will be described in details later in this section.

On the nature of the VM

The virtual machine's work consists in iterating over bytecode while executing specific code for each bytecode instruction. There are 109 bytecode instructions of fixed width. Some of them are followed by up to two bytes of arguments. Each module has a sequence of bytecode representing its toplevel forms and every function it contains. While iterating over a module's bytecode, the virtual machine increments a counter named the *program counter* which points to the index of the next instruction to execute. For example, if the VM reads a bytecode instruction which uses one byte argument, it will increase the program counter by two before executing the instruction.
The virtual machine uses a stack to save values which will be later used by bytecode instructions, or to save state at a given point of execution. Values are pushed on top of this stack, or *popped* (the last pushed value is read and removed) from the stack in a *last in first out* manner. The stack can contain any value of any size, all of which take one slot in the stack. It is indexed starting at zero going up, and the first slot available in the stack (at the top of the stack) is stored in the *stack index*. The stack is also used to push *stack frames* when invoking some function. The index of the top of the last stack frame is stored in the *stack pointer*. Stack frames and the stack pointer are described later in 4.2.6.
Aside from the stack, some instructions operate on specific structures and their members (the *current escaper* of a *thread* for example), or on a few general-purpose registers.

Threads

Since every code in ULM is executed by a thread, it follows that the virtual machine has a current thread variable, which holds all the information specific to a thread. That is, the information which needs to be saved by the thread when it is not scheduled, such as these members:

- A stack (%STACK),
- a stack pointer (%SP),
- a stack index (%SI),
- a program counter (%PC),
- a current module (%MOD),

- a current function (%FUN),
- a current environment (%ENV),
- a current escaper (%ESC),
- a current escaper's return value (%ESCRET),
- a current protector (%PROT),
- a current context (%CTX).

Virtual Machine

The virtual machine contains the following members:

- A list of loaded modules
- Two general-purpose registers (%VAL and %ARG)
- A current thread
- The scheduler

On our notations

When presenting a bytecode instruction, we use the following format:

`(FOO x y):  N`
Does some things.

Here we described that the instruction number `N` is named `FOO`, takes two byte arguments `x` and `y` and whose execution by the VM consists in doing some things.
In the remainder of this section we will present elements of compilation to and execution of these bytecodes. It is not our goal to describe the whole compiler of virtual machine formally, but for some elements of compilation we found it useful to use a notation similar to the denotational semantics notation we used earlier like such:

$\mathcal{C}[\![(\texttt{foo}\ \pi_1\ \pi_2)]\!]\rho\mathrm{t} =$
`(FOO 2 3)`$\S(\mathcal{C}[\![\pi_1]\!]\ \rho\ \mathrm{false})\S(\mathcal{C}[\![\pi_2]\!]\ \rho\ \mathrm{false})$

Which denotes that the `foo` form in ULM takes two arguments π_1 and π_2 and compiles it to a sequence of bytecode containing the `FOO` instruction with `2` and `3` as its arguments, followed by the compilation of π_1 and that of π_2. The compiler function named $\mathcal{C}$ takes a local environment ρ containing a list of lists of lambda-variables, and a boolean `t` representing the fact that an instruction is placed in tail-rec position or not. There are, of course, other arguments used by the real ULM compiler, such as the current module, but we have chosen to ignore them here for this informal description of the compiler.

4.2.2 Constants

There are several predefined constants in Scheme: the boolean constants `#t` and `#f`, the empty list `'()`, numbers, characters and symbols. These are the immutable values in Scheme. There are several mutable values which can appear in the source code, and are then treated as constants which cannot be muted. These include strings, vectors and pairs as introduced by the `(quote ...)` form.
All constant references affect the VAL register with the constant. There are a number of bytecodes allocated for predefined constants:

`(CONSTANT_TRUE)`: 10
    %VAL ← `#t`.

`(CONSTANT_FALSE)`: 11
    %VAL ← `#f`.

`(CONSTANT_NIL)`: 12
    %VAL ← `'()`.

`(CONSTANT_-1)`: 80
    %VAL ← `-1`

`(CONSTANT_0)`: 81
    %VAL ← `0`.

`(CONSTANT_1)`: 82
    %VAL ← `1`.

`(CONSTANT_2)`: 83
    %VAL ← `2`.

`(CONSTANT_4)`: 84
    %VAL ← `4`.

Other numbers between 0 and 255 are compiled to:

`(SHORT-NUMBER x)`: 79
    %VAL ← `x`.

Other numbers outside of -1 to 255, as well as any other constant are added to the module's list of constants, assigned a number and compiled to:

`(CONSTANT x)`: 9
    %VAL ← %CONSTANTS[`x`].

4.2.3 Variable reference

In order to access a variable, we need to know which kind of variable it is. In ULM we have local, global, imported and predefined variables, by order of lookup. Local variables represent the lexical environment, which we refer to from now on simply as the environment, stored in the current thread. The environment is represented by an ordered list of variables, and a pointer to the parent environment. The toplevel's environment is empty and has no parent.
If the variable we are accessing is a local variable belonging to the current environment, it is compiled to:

`(SHALLOW-ARGUMENT-REF0):` `1`

`(SHALLOW-ARGUMENT-REF1):` `2`

`(SHALLOW-ARGUMENT-REF2):` `3`

`(SHALLOW-ARGUMENT-REF3):` `4`

`(SHALLOW-ARGUMENT-REF j):` `5`
%VAL ← value of %ENV's first to `j`th variable.

If the variable we are accessing is a local variable belonging to a parent environment, it is compiled to:

`(DEEP-ARGUMENT-REF i j):` `6`
%VAL ← value of the `j`th variable of %ENV's `i`th parent environment.

Global variables refer to non-imported variables defined at the toplevel environment with `(define ...)`. They are indexed by number within the current module. A reference to a global variable from the current module is compiled to:

`(GLOBAL-REF i):` `7`
%VAL ← %GLOBALS[`i`]

Global variables imported from other modules are referred to as *imported variables*. They are referenced using the variable's name and module name rather than by index, in order to support separated compilation, since the imported module can be recompiled and the global indexes altered. The variable's name and its module's names are compiled in a constant of type `ImportedField`, added to the current module's constant pool and it is compiled as such:

`(IMPORTED-REF i j):` `6`
%VAL ← value of the variable represented by the `ImportedField` stored as the %CONSTANTS[`i + (256 * k)`]. If the imported variable's module is not loaded yet, it is loaded and initialised by the VM.

Predefined variables (the VM primitives) are referenced either by a few shortcut bytecodes, or by name:

`(PREDEFINED-CONS): 13`
%VAL ← the `cons` primitive.

`(PREDEFINED-CAR): 14`
%VAL ← the `car` primitive.

`(PREDEFINED-CDR): 15`
%VAL ← the `cdr` primitive.

`(PREDEFINED-PAIR?): 16`
%VAL ← the `pair?` primitive.

`(PREDEFINED-SYMBOL?): 17`
%VAL ← the `symbol?` primitive.

`(PREDEFINED-EQ?): 18`
%VAL ← the `eq?` primitive.

`(NPREDEFINED x): 41`
%VAL ← the primitive named by %CONSTANTS[`x`].

4.2.4 Variable affectation

Variable affectation is much simpler than reference, since only local and global variables are mutable. By design, imported variables and primitives are immutable.
If the variable we are affecting is a local variable belonging to the current environment, it is compiled to:

`(SET-SHALLOW-ARGUMENT!0): 21`

`(SET-SHALLOW-ARGUMENT!1): 22`

`(SET-SHALLOW-ARGUMENT!2): 23`

`(SET-SHALLOW-ARGUMENT!3): 24`

`(SET-SHALLOW-ARGUMENT! j): 25`
Sets the %ENV's first to `j`th variable to %VAL.

If the variable we are affecting is a local variable belonging to a parent environment, it is compiled to:

`(SET-DEEP-ARGUMENT! i j): 26`
Sets the `j`th variable of %ENV's `i`th parent environment to %VAL.

If we are affecting a global variable (of the current module), it is compiled to:

`(SET-GLOBAL! i): 27`
%GLOBALS[`i`] ← %VAL.

4.2.5 Conditional

In order to compile a conditional, we compile the condition π_1 to σ_1 and the *else* branch π_3 into σ_3. We then compile the *then* branch π_2 into σ_2, and append a `JUMP` instruction so that at the end of this branch we skip the *else* branch unconditionally. The result of the compilation is then the test σ_1 followed by a `JUMP-FALSE` conditional jump which skips the *then* branch if VAL is false, followed by the *then* branch σ_2 and the *else* branch σ_3.
Notice that all three expressions are compiled with the same environment. As for the tail-recursive information, the test π_1 is not tail-recursive, while both *then* and *else* branches are tailrec if the conditional is.

$\mathcal{C}[\![(\texttt{if}\ \pi_1\ \pi_2\ \pi_3)]\!]\rho\mathrm{t} =$
$\quad \underline{\texttt{let}}\ \sigma_1 = (\mathcal{C}[\![\pi_1]\!]\ \rho\ \mathrm{false}),\ \sigma_3 = (\mathcal{C}[\![\pi_3]\!]\ \rho\ \mathrm{t}),\ \sigma_2 = (\mathcal{C}[\![\pi_2]\!]\ \rho\ \mathrm{t})\S(\mathit{goto}\ \#\sigma_3)\ \underline{\texttt{in}}$
$\qquad \sigma_1\S(\mathit{jump\text{-}false}\ \#\sigma_2)\S\sigma_2\S\sigma_3$

$\mathit{goto} = \lambda\mathrm{n.n}{<}256 \rightarrow$ `(SHORT-GOTO n)`, `(LONG-GOTO n%256 n/256)`

$\mathit{jump\text{-}false} = \lambda\mathrm{n.n}{<}256 \rightarrow$ `(SHORT-JUMP-FALSE n)`, `(LONG-JUMP-FALSE n%256 n/256)`

The short jump is for jumps of less than 256 bytecodes:

`(SHORT-GOTO i)`: 30
$\quad \%\mathrm{PC} \leftarrow \%\mathrm{PC} +$ `i`.

The long jump is for jumps of more than 255 bytecodes:

`(LONG-GOTO i j)`: 28
$\quad \%\mathrm{PC} \leftarrow \%\mathrm{PC} +$ `i + (256 * j)`.

The short conditional jump is for conditional jumps of less than 256 bytecodes:

`(SHORT-JUMP-FALSE i)`: 31
$\quad \%\mathrm{PC} \leftarrow \%\mathrm{PC} +$ `i` $\underline{\text{iff}}\ \%\mathrm{VAL} =$ `#f`.

The long jump is for jumps of more than 255 bytecodes:

`(LONG-JUMP-FALSE i j)`: 29
$\quad \%\mathrm{PC} \leftarrow \%\mathrm{PC} +$ `i + (256 * j)` $\underline{\text{iff}}\ \%\mathrm{VAL} =$ `#f`.

4.2.6 Invocation

In order to invoke a function, we need a structure to hold the argument values that the caller is going to give the function. We call this structure the *activation frame*. It is in fact an array which holds the argument values, and has a free slot which will contain a link to the next activation frame. This link is not set at the invocation site, but within the closure as we will see in 4.2.7. This linked list of activation frames is the lexical environment of local variables.

Compiling an application is straightforward: we compile the function π_λ and push the result to the stack with `PUSH-VALUE` for later use. We do the same for each argument value, then when each argument has been pushed to the stack, we create an activation frame of the correct size into VAL, and pop each argument value from the stack to their corresponding place in the activation frame. This is done by the compiler function *compile-args*. Once all argument values have been popped from the stack, we pop the function in the current thread's current function with `POP-FUNCTION`.
At this point the application compilation differs depending on the tail-recursive status t: if this is a terminal call, we only need to jump to the function with `FUNCTION-GOTO`. Otherwise the execution will have to come back to the current function after the function call, which means that we have to save enough data so that the current thread can return to this function after the call we are compiling. In order to come back, we need to save the current environment, the PC, the current module and the current function[4]. This information (except the environment for implementation reasons) is stored in the stack in *stack frames*. A stack frame in ULM consists in the PC, function, module and pointer to the previous stack frame, called the *stack pointer*. At every non tail-rec invocation we push a stack frame and update the current thread's stack pointer to point to the newest frame.
For non tail-rec invocations, we store the environment in the stack with `PRESERVE-ENV`, then invoke the function with `FUNCTION-INVOKE` (which pushes a new stack frame), and when we return from this function we restore the environment by popping it from the stack with `RESTORE-ENV`. The compilation is thus as follows:

$$\begin{aligned}
&\mathcal{C}[\![(\pi_\lambda\ \pi_1\ \ldots\ \pi_n)]\!]\rho t = \\
&\quad \underline{\texttt{let*}}\ \sigma_\lambda = (\mathcal{C}[\![\pi_\lambda]\!]\ \rho\ \text{false}),\ \sigma_{\text{args}} = (\textit{compile-args}\ (\pi_1\ \ldots\ \pi_n)\ 0\ \rho), \\
&\qquad \sigma_{\text{app}} = \sigma_\lambda \S \texttt{(PUSH-VALUE)} \S \sigma_{\text{args}} \S \texttt{(POP-FUNCTION)}\ \underline{\text{in}} \\
&\qquad t \rightarrow \sigma_{\text{app}} \S \texttt{(FUNCTION-GOTO)}, \\
&\qquad \sigma_{\text{app}} \S \texttt{(PRESERVE-ENV)} \S \texttt{(FUNCTION-INVOKE)} \S \texttt{(RESTORE-ENV)}
\end{aligned}$$

$$\begin{aligned}
&\textit{compile-args} = \lambda \text{Ln}\rho.\text{L}{=}\emptyset \rightarrow (\textit{allocate-frame}\ \text{n}), \\
&\qquad (\mathcal{C}[\![\text{L}{\downarrow_1}]\!]\ \rho\ \text{false}) \S \texttt{(PUSH-VALUE)} \S (\textit{compile-args}\ \text{L}{\dagger}1\ \text{n}{+}1\ \rho) \S (\textit{pop-frame!}\ \text{n})
\end{aligned}$$

$$\begin{aligned}
&\textit{allocate-frame} = \lambda \text{n}.(=\text{n}\ 0) \rightarrow \texttt{(ALLOCATE-FRAME}_0\texttt{)}, (=\text{n}\ 1) \rightarrow \texttt{(ALLOCATE-FRAME}_1\texttt{)}, \\
&\qquad (=\text{n}\ 2) \rightarrow \texttt{(ALLOCATE-FRAME}_2\texttt{)}, (=\text{n}\ 3) \rightarrow \texttt{(ALLOCATE-FRAME}_3\texttt{)}, \\
&\qquad \texttt{(ALLOCATE-FRAME n)}
\end{aligned}$$

$$\begin{aligned}
&\textit{pop-frame!} = \lambda \text{n}.(=\text{n}\ 0) \rightarrow \texttt{(POP-FRAME!}_0\texttt{)}, (=\text{n}\ 1) \rightarrow \texttt{(POP-FRAME!}_1\texttt{)}, \\
&\qquad (=\text{n}\ 2) \rightarrow \texttt{(POP-FRAME!}_2\texttt{)}, (=\text{n}\ 3) \rightarrow \texttt{(POP-FRAME!}_3\texttt{)}, \texttt{(POP-FRAME! n)}
\end{aligned}$$

Here are the relevant bytecodes:

```
(PUSH-VALUE): 34
```

[4]The trained reader will wonder why we need so save the module if it can be obtained by the current function, since functions belong to a module. This is true, but fails for the toplevel, where there is a current module but no current function.

%STACK[%SI] ← %VAL.
%SI ← %SI + 1.

`(ALLOCATE-FRAME0):` 50

`(ALLOCATE-FRAME1):` 51

`(ALLOCATE-FRAME2):` 52

`(ALLOCATE-FRAME3):` 53

`(ALLOCATE-FRAME n):` 55
%VAL ← a new activation frame for zero to `n` argument values.

`(POP-FRAME!0):` 60

`(POP-FRAME!1):` 61

`(POP-FRAME!2):` 62

`(POP-FRAME!3):` 63

`(POP-FRAME! n):` 64
Pops the last value from the stack into the first to `n`th argument value slot in the activation frame located in %VAL.
%SI ← %SI - 1.

`(POP-FUNCTION):` 39
%SI ← %SI - 1.
%FUN ← %STACK[%SI].

`(PRESERVE-ENV):` 37
%STACK[%SI] ← %ENV.
%SI ← %SI + 1.

`(RESTORE-ENV):` 38
%SI ← %SI - 1.
%ENV ← %STACK[%SI].

The nature of what happens during invocation differs depending on what we are invoking. Invoking a closure consists in installing its captured environment to the current thread's environment, and setting the PC and current module to the closure's bytecode start and its module. Invoking a primitive consists usually[5] in executing the primitive's code in the VM, storing the returned value in VAL and calling the equivalent of the `RETURN` bytecode. The last type of object it is possible to invoke is the *escaper* functions given by `bind-exit`, they are described in 4.2.10.
All invocations are triggered by the same bytecodes:

[5]Some rare primitives behave differently, such as those which need to cooperate, described in 5.1.5.

`(FUNCTION-INVOKE): 45`

For a closure invocation:
%STACK[%SI] ← %PC.
%STACK[%SI+1] ← %FUN.
%STACK[%SI+2] ← %MOD.
%STACK[%SI+3] ← %SP.
%SI ← %SI + 4.
%SP ← %SI.
%ENV ← %FUN.ENV.
%MOD ← %FUN.MOD.
%PC ← %FUN.PC.
For a primitive: %VAL ← result of applying %FUN.
For an escaper, see 4.2.10.

`(FUNCTION-GOTO): 46`

For a closure invocation:
%ENV ← %FUN.ENV.
%MOD ← %FUN.MOD.
%PC ← %FUN.PC.
For a primitive: %VAL ← result of applying %FUN.
For an escaper, see 4.2.10.

Here is an example of primitive invocation and imported function tail-rec invocation:

```
(module bla
 (import r5rs))

(display 4)
(list 1 2 3)
```

Which would be compiled to the following:

```
Constant[0]: display
Constant[1]: ImportedField module: r5rs, variable: list
Bytecode[0]:(NPREDEFINED 0) [display]
Bytecode[2]:(PUSH-VALUE)
Bytecode[3]:(CONSTANT_4)
Bytecode[4]:(PUSH-VALUE)
Bytecode[5]:(ALLOCATE-FRAME1)
Bytecode[6]:(POP-FRAME!0)
Bytecode[7]:(POP-FUNCTION)
Bytecode[8]:(PRESERVE-ENV)
Bytecode[9]:(FUNCTION-INVOKE)
Bytecode[10]:(RESTORE-ENV)
Bytecode[11]:(IMPORTED-REF 1 0) [r5rs:list]
Bytecode[14]:(PUSH-VALUE)
Bytecode[15]:(CONSTANT_1)
Bytecode[16]:(PUSH-VALUE)
Bytecode[17]:(CONSTANT_2)
Bytecode[18]:(PUSH-VALUE)
Bytecode[19]:(SHORT-NUMBER 3)
Bytecode[21]:(PUSH-VALUE)
Bytecode[22]:(ALLOCATE-FRAME3) [stack contains: list 1 2 3]
Bytecode[23]:(POP-FRAME!2)
Bytecode[24]:(POP-FRAME!1)
Bytecode[25]:(POP-FRAME!0)
Bytecode[26]:(POP-FUNCTION)
Bytecode[27]:(FUNCTION-GOTO)
```

4.2.7 Abstraction

Lambdas are compiled and then assigned a number and stored in the module's list of function descriptors. The code of the lambda itself is compiled as follows for fixed arguments: we make a list of the argument names ν_1 to ν_n into θ, which we prepend to the current environment ρ into the environment ρ_1. We then compile the lambda's code π into σ_π with the environment ρ_1.
We have seen that lambda applications build an unlinked activation frame with the argument values in VAL. The lambda's code is then a check verifying the correct arity of that activation frame with `ARITY=?`, an instruction which extends the captured environment with the argument values, the body σ_π followed by the `RETURN` instruction, which returns to the caller.
Once that code is assigned a number and a slot in the current module's function descriptor list with the compiler function *store-closure*, the compilation of the lambda creation is the `CREATE-CLOSURE` bytecode with the number of the function descriptor.

$\mathcal{C}[\![(\lambda\ (\nu_1\ \ldots\ \nu_n)\ \pi)]\!]\rho t =$
  `let*` $\theta = (\nu_1\ \ldots\ \nu_n)$, $\rho_1 = \theta \S \rho$, $\sigma_\pi = (\mathcal{C}[\![\pi]\!]\ \rho_1\ \text{true})$,
    $\sigma_\lambda = (\mathit{arity{=}?}\ \text{n}) \S$`(EXTEND-ENV)`$\S \sigma_\pi \S$`(RETURN)` `in`
    $(\mathit{create\text{-}closure}\ (\mathit{store\text{-}closure}\ \sigma_\lambda))$

arity=? $= \lambda$n.(= n 0) $\rightarrow$ `(ARITY=?0)`, (= n 1) $\rightarrow$ `(ARITY=?1)`,
(= n 2) $\rightarrow$ `(ARITY=?2)`, (= n 3) $\rightarrow$ `(ARITY=?3)`, `(ARITY=? n)`

create-closure $= \lambda$n.`(CREATE-CLOSURE n%256 n/256)`

As for variable arity closures, there are a few differences: the arity is checked with `ARITY>=?`, which requires that at least all fixed arguments are given during invocation, then all additional arguments are packed into a list as the last argument's value with `PACK-FRAME!`.

$\mathcal{C}[\![(\lambda\ (\nu_1\ \ldots\ \nu_n\ .\ \nu_r)\ \pi)]\!]\rho t =$
$\underline{\texttt{let*}}\ \theta = (\nu_1\ \ldots\ \nu_n\ \nu_r),\ \rho_1 = \theta\S\rho,\ \sigma_\pi = (\mathcal{C}[\![\pi]\!]\ \rho_1\ \text{true}),$
$\sigma_\lambda =$ `(ARITY>=? n)§(PACK-FRAME! r)§(EXTEND-ENV)§`σ_π`§(RETURN)` $\underline{\texttt{in}}$
(*create-closure* (*store-closure* σ_λ))

`(ARITY=?0):` 71

`(ARITY=?1):` 72

`(ARITY=?2):` 73

`(ARITY=?3):` 74

`(ARITY=? x):` 75
Throws an exception unless the activation frame in %VAL has zero to `x` arguments.

`(ARITY>=? x):` 76
Throws an exception unless the activation frame in %VAL has `x` or more arguments.

`(PACK-FRAME! x):` 44
Replaces the `x`th slot of the activation frame in %VAL with a list containing all the values from the slots `x` and up. This has the effect of taking all optional arguments given to the invocation and packing them into a list as the lambda's last argument.

`(EXTEND-ENV):` 76
Takes the activation frame stored in %VAL, assigns its *next* link to %ENV, then stores the new activation frame in %ENV.

`(RETURN):` 43
%SP ← %STACK[%SI-1].
%FUN ← %STACK[%SI-2].
%MOD ← %STACK[%SI-3].
%PC ← %STACK[%SI-4].
%SI ← %SI - 4.

`(CREATE-CLOSURE i j):  40`
: %VAL ← a new closure instance with the function descriptor in %CONSTANTS[`i + (j * 256)`], capturing %ENV.

Here is an example of lambda abstraction:

```
(module bla)

(define (foo x)
 x
 (lambda y x))
```

Which would be compiled to the following (which is explained below):

```
FunctionDescriptor[0]: start: 6, length: 7
FunctionDescriptor[1]: start: 13, length: 9
Bytecode[0]:(CREATE-CLOSURE 0 0)
Bytecode[3]:(GLOBAL-SET! 0)       [foo]
Bytecode[5]:(RETURN)
Bytecode[6]:(ARITY=?1)
Bytecode[7]:(EXTEND-ENV)
Bytecode[8]:(SHALLOW-ARGUMENT-REF0) [x]
Bytecode[9]:(CREATE-CLOSURE 1 0)
Bytecode[12]:(RETURN)
Bytecode[13]:(ARITY>=? 0)
Bytecode[15]:(PACK-FRAME! 0)
Bytecode[17]:(EXTEND-ENV)
Bytecode[18]:(DEEP-ARGUMENT-REF 1 0) [x]
Bytecode[21]:(RETURN)
```

This example illustrates not only lambda variable references, but also the fact that there is an extra `RETURN` bytecode added at the end of each module's toplevel. We did not present it earlier in our examples because we did not define this bytecode until now, but every future example will contain it. It is placed at the end of the toplevel bytecode in order for the VM to both mark the end and stop executing the bytecode (otherwise the VM would start running the first function's bytecode), but also to return to the instruction which triggered the module's loading and toplevel execution.

In fact, there is also a `FINISH` bytecode prepended to each module's bytecode. When the main module is executed by the VM, we push an initial stack frame which returns to the `FINISH` instruction. When the VM reaches this instruction it means the main module's toplevel is done, and the implicit thread should be terminated, and the next one should be scheduled.

4.2.8 let, let* and letrec

The successive local variable declaration form `let*` is expanded prior to compilation to its equivalent `let` form as such:

```
1: (let* ((name1 value1)
2:        ...
3:        (nameN valueN))
4:  expr)
```

```
1: (let ((name1 value1))
2:  ...
3:  (let ((nameN valueN))
4:   expr))
```

The recursive local variable declaration form `letrec` is also expanded prior to compilation to its equivalent `let` form as such:

```
1: (letrec ((name1 value1)
2:          ...
3:          (nameN valueN))
4:  expr)
```

```
1: (let ((gensym1 #unspecified)
2:       ...
3:       (gensymN #unspecified))
4:  (set! gensym1 value1)
5:  ...
6:  (set! gensymN valueN)
7:  (let ((name1 gensym1)
8:        ...
9:        (nameN gensymN))
10:  expr))
```

Named `let` forms are also expanded prior to compilation to its equivalent unnamed `let` form:

```
1: (let loop ((name1 value1)
2:            ...
3:            (nameN valueN))
4:  expr)
```

```
1: (let ((loop #unspecified))
2:  (set! loop
3:   (lambda (name1 ... nameN)
4:    expr))
5:  (loop value1 ... valueN))
```

The unnamed `let` form is then compiled similarly to a combination of application and abstraction, but without the expense of building a closure: we evaluate all the variable values into a new environment as we would do for the arguments given to a function, then we extend the current environment to evalue the body. If the `let` is not tail-rec we un-extend the environment after the body with `UNLINK-ENV`:

$\mathcal{C}[\![$`(let ((`ν_1 π_1`) ... (`ν_n π_n`)) `π`)`$]\!]\rho$t =
  `let*` σ_{args} = (*compile-args* (π_1 ... π_n) 0 ρ), ρ_1 = (ν_1 ... ν_n)§ρ,
    σ_{body} = ($\mathcal{C}[\![\pi]\!]$ ρ t) `in`
    t → σ_{args}§`(EXTEND-ENV)`§σ_{body},
    σ_{args}§`(EXTEND-ENV)`§σ_{body}§`(UNLINK-ENV)`

`(UNLINK-ENV): 33`
  %ENV ← %ENV.next.

Here is an example of local variable declaration:

```
(module bla)

(let ((foo 2))
 foo)
```

Which would be compiled to the following:

```
Bytecode[0]:(CONSTANT_2)
Bytecode[1]:(PUSH-VALUE)
Bytecode[2]:(ALLOCATE-FRAME1)
Bytecode[3]:(POP-FRAME!0)
Bytecode[4]:(POP-FUNCTION)
Bytecode[5]:(EXTEND-ENV)
Bytecode[6]:(SHALLOW-ARGUMENT-REF0)
Bytecode[7]:(RETURN)
```

4.2.9 Protection

The protection introduced by (unwind-protect π_1 π_2) is the first primitive introducing the *dynamic environment* in ULM.
The protection block specify that when π_1 is evaluated, π_2 is to be evaluated before leaving the protection block, either after normal termination of π_1 or its preemption. In the case of normal termination, `unwind-protect` returns the value returned by π_1 after π_2 is evaluated. In the case of preemption, `unwind-protect` does not return any value since the preemption's unwinding of the stack is resumed after π_2 is evaluated.
In order to be able to return to the evaluation of π_2 in the case of preemption, we need to store some information in the protector structure. This includes the equivalent of a stack frame: the stack pointer, the PC of the protection code, its module and function, the current environment as well as the stack index (since the stack can be unwound by preemption). We also store the next protector, and the eventual value returned by π_1.
Protection blocks are represented at runtime by a linked list of protector structures stored in the current thread's current protector. Each time we enter a protection block we create a new current protector in which we store the previous protector. When we leave a protection block we remove the current protector and replace it with its previous protector. Thus at any time the dynamic list of protectors is accessible through the current protector.
The compilation is then straightforward: we compile π_1 and π_2 into σ_1 and σ_2, then the compiled form consists in pushing a new protector for the code located after σ_1 with `PUSH-PROTECTOR`, executing σ_1, invalidate the current protector, store the returned value and restore the previous protector with `INVALIDATE-PROTECTOR`, then execute σ_2 before either returning the stored value or resume preemption with `POP-PROTECTOR`:

$\mathcal{C}[\![$(unwind-protect π_1 π_2)$]\!]\rho$t =
  let* $\sigma_1 = (\mathcal{C}[\![\pi_1]\!]\ \rho$ false), $\sigma_2 = (\mathcal{C}[\![\pi_2]\!]\ \rho$ false) in
    (PUSH-PROTECTOR #σ_1)§σ_1§(INVALIDATE-PROTECTOR)§σ_2§(POP-PROTECTOR)

(PUSH-PROTECTOR x): 253
  %PROT ← a new protector.
  %PROT.next ← the previous %PROT.
  %PROT.SI ← %SI.
  %PROT.SP ← %SP.
  %PROT.PC ← %PC + x (representing the start of σ_2).
  %PROT.ENV ← %ENV.

%PROT.MOD ← %MOD.
%PROT.FUN ← %FUN.
%STACK[%SI] ← %PROT.
%SI ← %SI + 1.

`(INVALIDATE-PROTECTOR): 254`
%PROT.VAL ← %VAL.
%PROT ← %PROT.next.

`(POP-PROTECTOR): 252`
%SI ← %SI - 1.
If we are in the course of stack unwinding, resume it.
Otherwise %VAL ← %STACK[SI].VAL.

4.2.10 Strong preemption

Strong preemption with (`bind-exit` (ν) π) makes it possible to preempt the execution of π within its dynamic extent by calling the function ν. As was the case with `unwind-protect` we need to be able to return from `bind-exit` through preemption, and so we have to store enough state to do so. This is done with `PUSH-ESCAPER` by pushing the environment, an escaper structure and a stack frame (with PC pointing after π) on the stack. The environment of π is then augmented with that escaper structure as ν.
This escaper structure contains a stack index equal to the stack pointer we just updated. The compilation of π is followed by `RETURN`, in order to pop the stack frame we created, and `POP-ESCAPER` which pops the escaper from the stack and restores the environment. The compilation is then as follows:

$\mathcal{C}[\![$(`bind-exit` (ν) π)$]\!]\rho$t =
  `let` σ = ($\mathcal{C}[\![\pi]\!]$ (ν)§ρ true) `in` `(PUSH-ESCAPER #`σ`)`§σ§`(RETURN)`§`(POP-ESCAPER)`

The invocation of the escaper involves storing the escaper as the current escaper (%ESC), and its argument value as the escaper's return value (%ESCRET), then unwinding. Unwinding consists in invoking all the protectors whose stack index is greater than the escaper's, then restoring the escaper's stack index and calling the equivalent of `RETURN` to return to the `POP-ESCAPER` of the `bind-exit`.

`(PUSH-ESCAPER x): 251`
%STACK[%SI] ← %ENV.
%STACK[%SI + 1] ← a new escaper.
%STACK[%SI + 2] ← %PC + x.
%STACK[%SI + 3] ← %FUN.
%STACK[%SI + 4] ← %MOD.
%STACK[%SI + 5] ← %SP.
%STACK[%SI + 1].SI ← %SI + 6.
%ENV ← a new activation frame with one slot.
%ENV.next ← the previous %ENV.
%ENV[0] ← %STACK[%SI + 1].
%SI ← %SI + 6.
%SP ← %SI.

`(POP-ESCAPER): 250`
%ENV ← %STACK[%SI - 2].
%SI ← %SI - 2.

4.2.11 Miscellaneous bytecodes

There are several Scheme primitives which are often called. For these we have assigned a few bytecodes to speed up the calls. There is one primitive taking zero arguments:

`(CALL0-newline): 88`
%VAL ← `(linebreak)`.

Then there are several primitive calls taking one argument which the compiler stores in the %VAL register:

`(CALL1-car): 90`
%VAL ← `(car %VAL)`.

`(CALL1-cdr): 91`
%VAL ← `(cdr %VAL)`.

`(CALL1-pair?): 92`
%VAL ← `(pair? %VAL)`.

`(CALL1-symbol?): 93`
%VAL ← `(symbol? %VAL)`.

`(CALL1-display): 94`
%VAL ← `(display %VAL)`.

`(CALL1-print): 95`
%VAL ← `(print %VAL)`.

`(CALL1-null?): 96`
%VAL ← `(null? %VAL)`.

`(CALL1-eof-object?): 98`
%VAL ← `(eof-object? %VAL)`.

Then there are several primitive calls taking two arguments. For these the compiler stores their first argument in %ARG and their second in %VAL:

`(POP-ARG): 35`
%SI ← %SI - 1.
%ARG ← %STACK[%SI].

`(CALL2-cons): 100`
%VAL ← `(cons %ARG %VAL)`.

`(CALL2-eq?): 101`
%VAL ← `(eq? %ARG %VAL)`.

`(CALL2-set-car!): 102`
%VAL ← `(set-car! %ARG %VAL)`.

`(CALL2-set-cdr!): 103`
%VAL ← `(set-cdr! %ARG %VAL)`.

`(CALL2-+): 104`
%VAL ← `(+ %ARG %VAL)`.

`(CALL2--): 105`
%VAL ← `(- %ARG %VAL)`.

`(CALL2-=): 106`
%VAL ← `(= %ARG %VAL)`.

`(CALL2-<): 107`
%VAL ← `(< %ARG %VAL)`.

`(CALL2->): 108`
%VAL ← `(> %ARG %VAL)`.

`(CALL2-*): 109`
%VAL ← `(* %ARG %VAL)`.

`(CALL2-<=): 110`
%VAL ← `(<= %ARG %VAL)`.

`(CALL2->=): 111`
%VAL ← `(>= %ARG %VAL)`.

`(CALL2-remainder): 112`
%VAL ← `(remainder %ARG %VAL)`.

When asked to do so, the compiler will include debugging information in the bytecode. This is represented by the `DEBUG` bytecode which is output by the compiler at various places to indicate that the following bytecodes originate from a given character index in the source code.

`(DEBUG x y): 255`
Tells the VM that the following bytecode originates from the character number `256 * y + x` in the module's source file.

`(FINISH): 20`
Terminate the implicit thread and return to the scheduler.

4.3 The OULM file format

ULM's bytecode is compiled into a file format called *OULM*. This format is based on Java's Class file format [LY99]. It was chosen instead of the SEXP-based format we used at first (taken from Queinnec's L.I.S.P. book [Que96]) at the time we implemented the J2ME VM in order to save space by choosing both a more compact format, and one that could be read in a Java VM without implementing a Scheme reader.

The following table describes the size in kilobytes of the whole ULM runtime as source ULM files, as well as compiled to the old (SEXP-based) OULM format and the new OULM format. The three sizes are shown in uncompressed and compressed form because our runtime has to be compressed into a JAR file (using ZIP compression) for distribution on J2ME devices. It is therefore relevant to compare both the uncompressed sizes for the Bigloo VM distribution as well as the compressed sizes required for the J2ME VM.

Version	Uncompressed	zip
Source ULM	61.1	27.4
Old OULM	53.6	10.9
New OULM	24.9	11.5

This table shows that the uncompressed ULM installation size is greatly reduced with the new bytecode format compared to both the old format and the source code. In zipped form the new format is still better than the source code, but it is slightly larger than the old format. This is not totally relevant because since the time we migrated to the new format we added a few features and fixed a few bugs which required a slight growth in the bytecode format size. On the other hand it could be argued that the SEXP format can be compressed by ZIP slightly more efficiently than our OULM format.

While in retrospect the Java Class file format is not optimal for storage when size matters (like on J2ME) [BHV98] [Pug99] [RMH99], it is a very simple format, well-known and described in great detail, and by choosing it as base we can benefit from many research done to improve it. In the end we did not have to compress the format further, by sharing the constant pools, using Huffman-encoded bytecodes [LF05] or even grouping bytecodes commonly grouped together, because we were left with enough room for our J2ME phone.

We will now describe the OULM bytecode format.

4.3.1 Overall Structure

Each ULM module is compiled to a single OULM file composed of several sections: `Header`, `Constants`, `ModuleInformation`, `Globals`, `FunctionDescriptors` and `Attributes`. All sizes are expressed in bytes, with a suffix of 's' for signed, 'u' for unsigned, 'f' for IEEE-754 encoded floating numbers and 'p' for a pointer to an entry in the constant pool. For example, '4u' represents a 4-byte unsigned number, and '8f' an 8-byte floating number. All data is stored in big-endian format.

The table in Figure 4.1 represents the various sections and fields of the OULM format, with their size and their expected value:

Section	Field	Size	Value
`Header`	`MagicNumber`	4u	0xfeedabee
	`MajorVersion`	2u	1
	`MinorVersion`	2u	5
`Constants`	`ConstantsCount`	2u	The number of Constant fields
	`Constants`	?	The Constant list
`ModuleInfo`	`ModuleName`	2p	Index of the ModuleName constant
	`Modifiers`	1u	Modifiers for the module
`GlobalVariables`	`GlobalsCount`	2u	Number of Global fields
	`Globals`	?	The Global list
`FunctionDescriptors`	`FunDescCount`	2u	Number of FunDesc fields
	`FunDescs`	?	The FunDesc list
`Attributes`	`AttributesCount`	2u	Number of Attribute fields
	`Attributes`	?	The Attribute list

Table 4.1: OULM sections

4.3.2 Header

The `Header` section of the OULM format is composed of the `MagicNumber` field with the 0xfeedabee value, which is used to ensure the file is in the OULM format. After that we have two fields: `MajorVersion` and `MinorVersion` denoting the version of the OULM format for the rest of the file. The current version, which is the one we are describing here, is 1.5.

4.3.3 Constants

The `Constants` section is composed of the `ConstantsCount` field and an ordered list of `Constant` fields. The `ConstantsCount` field indicates the total number of `Constant` fields. Each Constant field starts with a 1u `ConstantType` field which denotes the type of constant being read. Each `Constant` field has a different format after the `ConstantType`, and is of variable length. The constants are referred to by number in the bytecode. The list of different `Constant` fields by type is shown in Figure 4.2.
Although Scheme does not usually constrain number representation sizes, we are bound to the Java and Bigloo languages of our virtual machines and their representation of numbers. J2ME does not provide the BigInteger, for unrestricted integer representation, and both Bigloo and Java require their integers and floats to be in 32 or 64 bits. As a consequence, our VM represents integers and floats in 32bits unless otherwise specified.
`Character`, `String`, `Symbol` and `Keyword` constants are represented in Unicode [Wes03], as specified by R^6RS. Except for the `Character` constant type which is represented by a 4u unicode index, the *string*-like constants are stored using a 2u `Length` field for the string's number of characters, and the string encoded in the UTF-8 character set, in which each Unicode character is stored in a variable-width format of one to four bytes.
The `Vector` constant data is stored using a 2u `Length` indicating the length of the vector, followed by as many 2p `Data` fields pointing to entries in the constant pool for each vector index. `Pair` constants are represented with two 2p fields `Car` and `Cdr` pointing to entries

Type	ConstantType	Data size	Data
`Integer`	0	4s	32bit signed integer
`Float`	1	4f	32bit IEEE-745 floating number
`Char`	2	4u	32bit unsigned unicode character
`Long`	3	8s	64bit signed integer
`Double`	4	4f	64bit IEEE-745 floating number
`String`	5	2u + ?	2u (Length) + UTF-8 encoded string
`Symbol`	6	2u + ?	2u (Length) + UTF-8 encoded string
`Keyword`	15	2u + ?	2u (Length) + UTF-8 encoded string
`Pair`	7	2p + 2p	2p (Car) + 2p (Cdr)
`Nil`	8	0	*No data needed for `'()`*
`True`	11	0	*No data needed for `#t`*
`False`	12	0	*No data needed for `#f`*
`Unspecified`	13	0	*No data needed for `#unspecified`*
`Vector`	14	2u + n*2p	2u (Length) + n*2p (Data)
`ImportedField`	9	2p + 2p	2p (ModuleName) + 2p (GlobalName)
`ExternField`	10	2p + 2p	2p (ModuleName) + 2p (GlobalName)

Table 4.2: OULM constants

in the constant pool.
`ImportedField` and `ExternField` represent extra-module global variable references and native module references as introduced by the `IMPORTED-REF` bytecode. They contain two 2p fields `ModuleName` and `VariableName` pointing to the constants holding the module and variable names, of the Symbol type.
The other `True`, `False`, `Nil` constants represent the `#t`, `#f`, and empty list values respectively, while the `Unspecified` constant type represents the `#unspecified` value which is used by some to return an unspecified value.

4.3.4 Module Information

The `ModuleInformation` section contains the `ModuleName` and `Modifiers` fields. The 2p `ModuleName` field is an index to a Symbol entry in the constant pool containing the name of the ULM module. The `Modifiers` field is a 1u bitset which can contain any binary disjunction of the following values:

`MOD_DEBUG: 1`
: Set if the present module contains debug information.

`MOD_UBIQUITOUS: 2`
: Set if the present module is ubiquitous.

4.3.5 Global Variables

The `GlobalsCount` field denotes the number of global variables in the current module. It is followed by an ordered list of `Global` fields. There are two types of `Global` fields: `GlobalVariable` and `GlobalMethod` representing respectively a variable of unknown type

and a variable containing a function of known prototype. Each Global field starts with a 1u `GlobalType` field denoting its type. Here is the list of the different `Global` fields by type:

Name	GlobalType	Data size	Data
`GlobalVariable`	0	1u + 2p	1u (`Mods`) + 2p (`Name`)
`GlobalMethod`	1	1u + 2p + 1u	1u (`Mods`) + 2p (`Name`) + 1u (`Arity`)

Each type of `Global` field contains a 1u `Mods` (modifiers) field and a 2p `Name` field referencing a Symbol constant containing its name. The `Mods` field contains a binary disjunction holding information about the variable. The following value is used as a modifier for both types of `Global` fields:

`MOD_EXPORTED: 1`
: Set if the global variable is exported.

The following modifiers are only relevant for the `GlobalMethod` type:

`MOD_DOTTED: 2`
: Set if the global function is of variable arity.

`MOD_MACRO: 4`
: Set if the global function is a macro.

The `GlobalMethod` field has an additional 1u `Arity` field representing its arity (in case of variable arity, it represents the minimum number of arguments accepted).

4.3.6 Function Descriptors

Each lambda present in the ULM module is assigned a function descriptor during compilation, used with the `CREATE-CLOSURE` bytecode. The `FunctionDescriptors` section is composed of a 2u `FunDescsCount` field containing the number of function descriptors, followed by an ordered list of `FunDesc` fields. Each `FuncDesc` field has the following components:

Name	Size	Contents
`CodeOffset`	4u	The index of the function's first bytecode
`CodeLength`	2u	The number of bytecodes for the function

4.3.7 Attributes

The `Attributes` section is composed of a 2u `AttributesCount` field containing the number of `Attribute` fields present in the module, followed by the list of `Attribute` fields. Each `Attribute` field is composed of a 2p `AttributeName` field containing an index to a String entry in the constant table, followed by a 4u `AttributeLength` field containing the length of the remaining `AttributeData` field.

AttributeName	AttributeLength	AttributeData
`Code`	?	The bytecodes
`Main`	2p	2p (`Name`)
`SourceFile`	2p	2p (`File`)
`Extern`	?	2p (`Name`) + 2u (`MappingCount`) + n (`Mapping`)
`Debug`	?	2u (`DebugNamesCount`) + n (`DebugNames`)

Table 4.3: OULM attributes

There are five predefined `AttributeName` values: `Code`, `Main`, `SourceFile`, `Extern` and `Debug`. Future versions of the OULM format as well as users of the OULM format may introduce new attributes as long as their `AttributeLength` field is correct. They will be skipped by VM loaders which do not understand the new attributes. Their descriptions are shown in Figure 4.3.
The `Code` Attribute contains the bytecode of the module in its `AttributeData`. The `Main` Attribute contains a reference to the module's main function name as a Symbol in the constant pool. The `SourceFile` Attribute contains a reference to the module's source file as a String entry in the constant pool.
There is an `Extern` Attribute per extern module declared in the module header. This is used to describe native modules. Each `Extern` Attribute contains a 2p `Name` field pointing to a Symbol representing its name in the constant pool, a 2u `MappingCount` field denoting the number of `Mapping` fields to follow, and a list of `Mapping` fields. Each `Mapping` field represents a mapping between a ULM VM type and a native object representing the native module we are describing. For example, the `"org.foo.Bar"` Java class may provide the `bar` native module for the Java ULM VM backend, while it may be provided by the `foobar` Bigloo module for the Bigloo ULM VM backend. Each `Mapping` field is thus composed of a 2p `Backend` field pointing to the name of the backend as a Symbol entry in the Constant Pool, and a 2p `MapsTo` field reference to the constant pool:

Name	Size	Contents
`Backend`	2p	The backend
`MapsTo`	2p	The mapping for this backend

The `Debug` Attribute is optional and contains the names of each local variable in the source code. It is composed of a 2u `DebugNamesCount` field denoting the number of total `DebugNames` fields, and a list of `DebugNames` fields. The `DebugNames` field starts with a 1u `DbgType` field representing the type of local binding we are describing. Here is the list of such types:

Name	DbgType	Data size	Data
`Closure`	0	2u + 2u + ?	2u (`FunDesc`) + 2u (`NamesCount`) + 2p*n (`Names`)
`Let`	1	4u + 2u + ?	4u (`Bytecode`) + 2u (`NamesCount`) + 2p*n (`Names`)

```
(ubiq-module foo
 (import r5rs)
 (export (fu)
         (bear . r))
 (extern (bar
          ((bigloo . bar) (java . "org.foo.Bar"))
           (gee a b)))
)

(define bla #f)

(define (fu)
 (gee '(2 . ()) #(3)))

(define (bear . r)
 (let ((v 2))
  (list? (cons v '()))))
```

Figure 4.1: Example ULM program

The difference between both types of `DebugType` lies in how they associate a list of local variable names to the local binding they represent. `Closure` are represented by a 2u `FunDesc` field, while `Let` entries point to a 4u `Bytecode` offset. Both entries then have a 2u `NamesCount` field denoting the number of local variables present in the binding, followed by an ordered list of 2p `Name` fields referencing the names of the variables as Symbols in the constant pool. `Let` and `Closure` bindings with no variables are not present in the `Debug` Attribute.

4.3.8 Example

In order to illustrate the OULM file format, we will dissect an example ULM module containing all the code needed to present every aspect of the format. The example is shown in Figure 4.1.
This module is named `foo`, imports the `r5rs` module, exports the `fu` and `bear` variables with their prototypes and uses a native module `bar` provided in Bigloo by the `bar` module and in Java by the `org.foo.Bar` class, from which we import the `gee` function. Then we define three global variables `bla`, `fu` and `bear`.
Using our compiler, we compile this module to the `foo.oulm` file, then we present the compiled form section by section. There is nothing surprising in the `Header` section:

Header

```
MagicNumber: 0xfeedabee
MinorVersion: 1
MajorVersion: 5
```

The `Constants` section contains several constants present in the code:

1. An `ExternField` representing the `gee` variable of the `bar` native module.
2. A `Cons` representing the `'(2 . '())` constant.
3. A `Vector` representing the `#(3)` constant.
4. An `ImportedField` representing the `list?` variable of the `r5rs` module.
5. Several other constants introduced by compilation, which are used in later sections.

Constants

```
ConstantsCount: 25
Constant[0]: ExternField: 4/5
Constant[1]: Cons: (6 . 7)
Constant[2]: Vector: #(8)
Constant[3]: ImportedField: 9/10
Constant[4]: Symbol: "bar"
Constant[5]: Symbol: "gee"
Constant[6]: Integer: 2
Constant[7]: Nil
Constant[8]: Integer: 3
Constant[9]: Symbol: "r5rs"
Constant[10]: Symbol: "list?"
Constant[11]: String: "Code"
Constant[12]: String: "SourceFile"
Constant[13]: String: "Extern"
Constant[14]: Symbol: "bigloo"
Constant[15]: Symbol: "java"
Constant[16]: String: "org.foo.Bar"
Constant[17]: String: "/home/stephane/src/ulm/foo.ulm"
Constant[18]: String: "Debug"
Constant[19]: Symbol: "r"
Constant[20]: Symbol: "v"
Constant[21]: Symbol: "foo"
Constant[22]: Symbol: "bla"
Constant[23]: Symbol: "fu"
Constant[24]: Symbol: "bear"
```

The `ModuleInfo` section contains the name of the module and information about debug and ubiquity:

ModuleInfo

```
ModuleName: 21 (foo)
Modifiers: 11 (MOD_DEBUG | MOD_UBIQUITOUS)
```

The `Globals` section contains the three global variables we defined:

Globals

```
GlobalsCount: 3
Global[0]: GlobalVariable
 Modifiers: 0
 Name: bla
Global[1]: GlobalMethod
 Modifiers: 1 (MOD_EXPORTED)
 Name: 23 (fu)
 Arity: 0
Global[2]: GlobalMethod
 Modifiers: 11 (MOD_EXPORTED | MOD_DOTTED)
 Name: 4 (bear)
 Arity: 0
```

Our source code's two lambdas are described in the `FunctionDescriptors` section:

FunctionDescriptors

```
FunDescsCount: 2
FunDesc[0]: [CodeOffset: 24, CodeLength: 24]
FunDesc[1]: [CodeOffset: 48, CodeLength: 34]
```

We have four `Attribute` fields. The first one is the `Code` Attribute containing the module's bytecode:

Attributes

```
AttributesCount: 4
Attribute[0]: [AttributeName: Code, AttributeLength: 82]
```

This first section contains the toplevel:

```
0:(FINISH)
1:(PREDEFINED-FALSE)
2:(DEBUG 137 0)               [char: 137]
5:(SET-GLOBAL! 0)             [bla]
7:(CREATE-CLOSURE 0 0)        [index: 0]
10:(DEBUG 154 0)              [char: 154]
13:(SET-GLOBAL! 1)            [fu]
15:(CREATE-CLOSURE 1 0)       [index: 1]
18:(DEBUG 186 0)              [char: 186]
21:(SET-GLOBAL! 2)            [bear]
23:(RETURN)
```

The following corresponds to the function `fu`:

```
24:(ARITY=?1)
25:(EXTEND-ENV)
26:(DEBUG 168 0)              [char: 168]
29:(IMPORTED-REF 0 0)         [bar::gee]
32:(PUSH-VALUE)
33:(CONSTANT 1)               [(2 . ())]
35:(PUSH-VALUE)
36:(CONSTANT 2)               [#(3)]
38:(PUSH-VALUE)
39:(ALLOCATE-FRAME3)
40:(POP-FRAME!1)
41:(POP-FRAME!0)
42:(DEBUG 168 0)              [char: 168]
45:(POP-FUNCTION)
46:(FUNCTION-GOTO)
47:(RETURN)
```

The following corresponds to the function `bear`:

```
48:(ARITY>=? 1)
50:(PACK-FRAME! 0)
52:(EXTEND-ENV)
53:(CONSTANT2)
54:(PUSH-VALUE)
55:(ALLOCATE-FRAME2)
56:(POP-FRAME!0)
57:(EXTEND-ENV)               [let]
58:(DEBUG 220 0)              [char: 220]
61:(IMPORTED-REF 3 0)         [r5rs::list?]
64:(PUSH-VALUE)
65:(SHALLOW-ARGUMENT-REF0)    [v]
66:(PUSH-VALUE)
67:(PREDEFINED-NIL)
68:(POP-ARG1)
69:(DEBUG 227 0)              [char: 227]
72:(CALL2-cons)
73:(PUSH-VALUE)
74:(ALLOCATE-FRAME2)
75:(POP-FRAME!0)
76:(DEBUG 220 0)              [char: 220]
79:(POP-FUNCTION)
80:(FUNCTION-GOTO)
81:(RETURN)
```

Then we have the `SourceFile` Attribute:

```
Attribute[1]: [AttributeName: SourceFile, AttributeLength: 2]
 File: "/home/stephane/src/ulm/foo.ulm"
```

Follows the `Extern` Attribute, telling us that the `bar` native module is provided in Bigloo with the Bigloo module `bar` and in Java with the class `org.foo.Bar`:

```
Attribute[2]: [AttributeName: Extern, AttributeLength: 12]
 Name: bar
 MappingCount: 2
  Mapping[0]: Backend: bigloo, MapsTo: bar
  Mapping[1]: Backend: java, MapsTo: "org.foo.Bar"
```

At last we have the `Debug` Attribute, telling us about the `r` variable of the `bear` function, and the `v` variable of the `let`:

```
Attribute[3]: [AttributeName: Debug, AttributeLength: 30]
DebugNamesCount: 2
 DebugName[0]: DebugNamesType: Closure, FunDesc: 1, NamesCount: 1
  Name[0]: r
 DebugName[0]: DebugNamesType: Let, Bytecode: 57, NamesCount: 1
  Name[0]: v
```

Chapter 5

Implementation: ULM

One of the universal rules of happiness is: always be wary of any helpful item that weighs less than its operating manual.

– Terry Pratchett

While the virtual machines, their bytecode and the OULM file format presented in the past chapter could be used to implement any Scheme language, they have been built from the start with the goal of hosting ULM primitives in Scheme. To that end, we have devised several additional bytecodes, extensions to the OULM file format, and have added support in the virtual machines for thread scheduling and migration.
In this chapter we present the compilation of ULM primitives into new bytecodes, then we will talk about how migration works, and finally how we achieved a new kind of scheduling.

5.1 ULM bytecodes

The compilation of ULM primitives into bytecodes is presented using the same techniques and notations as in Section 4.2 in the previous chapter.

5.1.1 Thread creation

Threads are represented as closures for practical reasons, although this is not visible to the programmer. When creating a thread with (`ulm:thread` π), a new function descriptor is allocated, containing the compilation of π followed by an instruction which is called when the thread terminates `DESTROY-THREAD`. The compilation of a thread creation is then `CREATE-THREAD` with the index of the function descriptor:

$\mathcal{C}[\![$(`ulm:thread` π)$]\!]\rho$t =
  `let*` σ = ($\mathcal{C}[\![\pi]\!]$ ρ true)§`(DESTROY-THREAD)`, n = (*store-closure* σ) `in`
    `(CREATE-THREAD n%256 n/256)`

`(CREATE-THREAD i j)`: 140
  Creates a thread whose code is represented by the function descriptor `i + (j * 256)`, by capturing %ENV. The thread will be scheduled later.

`(DESTROY-THREAD): 145`
Terminates the current thread and returns to the scheduler.

5.1.2 Agent creation

Agents are fairly similar to threads in their compilation, except that they take a name as argument and return it to the caller:

$\mathcal{C}[\![(\texttt{ulm:agent}\ (\nu)\ \pi)]\!]\rho\mathrm{t} =$
$\underline{\texttt{let*}}\ \rho_1 = (\nu)\S\rho,\ \sigma = (\mathcal{C}[\![\pi]\!]\ \rho_1\ \mathrm{true})\S\texttt{(DESTROY-THREAD)},\ \mathrm{n} = (\textit{store-closure}\ \sigma)\ \underline{\texttt{in}}$
`(CREATE-AGENT n%256 n/256)`

`(CREATE-AGENT i j): 140`
Creates an agent whose code is represented by the function descriptor `i + (j * 256)`, by capturing %ENV extended with an activation frame containing the agent's name as its only argument value.
%VAL ← that name.
he agent will be scheduled later.

5.1.3 Suspension

The suspension introduced by $(\texttt{ulm:when}\ \pi_1\ \pi_2)$ involves suspending the execution of π_2 at every instant until the signal π_1 is emitted. This is represented in the scheduler by the thread context's list of preemption and suspension clauses. On the stack a suspension context is represented using a special version of the protector structure: one that stores the current thread's context in order to restore it as we unwind the stack during preemption. Compilation of the suspension primitive consists in compiling the signal π_1 into σ_1, then entering the when context with `ENTER-WHEN` to execute π_2 compiled into σ_2, then leave the when context with `LEAVE-WHEN`:

$\mathcal{C}[\![(\texttt{ulm:when}\ \pi_1\ \pi_2)]\!]\rho\mathrm{t} =$
$(\mathcal{C}[\![\pi_1]\!]\ \rho\ \mathrm{false})\S\texttt{(ENTER-WHEN)}\S(\mathcal{C}[\![\pi_2]\!]\ \rho\ \mathrm{false})\S\texttt{(LEAVE-WHEN)}$

`(ENTER-WHEN): 141`
clause ← a new suspension clause.
clause.signal ← %VAL.
%PROT ← a new protector.
%PROT.SI ← %SI + 1.
%PROT.CTX ← %CTX.
%PROT.next ← the previous %PROT.
%STACK[%SI] ← %CTX.
%SI ← %SI + 1.
%CTX ← clause § %CTX.
Relinquishes control to the scheduler if the signal is not emitted.

`(LEAVE-WHEN): 142`
%CTX ← %STACK[%SI - 1].
%PROT ← %PROT.next.

5.1.4 Weak preemption

The weak preemption introduced by (`ulm:watch` π_1 π_2) involves preempting the execution of π_2 at the end of the instant where signal π_1 is emitted. Like suspension, this is represented in the scheduler by the thread context's list of preemption and suspension clauses. On the stack a suspension context is represented using a special version of the protector structure: one that stores the current thread's context in order to restore it as we unwind the stack during preemption. The escaping mechanism is then exactly the same as that of strong preemption.
Compilation of the weak preemption primitive consists in compiling the signal π_1 into σ_1, then entering the watch context with `ENTER-WATCH` to execute π_2 compiled into σ_2, then leave the watch context with `RETURN` and `LEAVE-WATCH`:

$\mathcal{C}[\![$(`ulm:watch` π_1 π_2)$]\!]\rho$t =
`let` $\sigma_1 = (\mathcal{C}[\![\pi_1]\!]\ \rho$ false), $\sigma_2 = (\mathcal{C}[\![\pi_2]\!]\ \rho$ false) `in`
σ_1§`(ENTER-WATCH #`σ_2`)`§σ_2§`(RETURN)`§`(LEAVE-WATCH)`

`(ENTER-WATCH x): 146`
%STACK[%SI] ← %ENV.
clause ← a new preemption clause.
clause.signal ← %VAL.
clause.ESC ← a new escaper.
clause.ESC.SI ← %SI + 7.
%PROT ← a new protector.
%PROT.SI ← %SI + 7.
%PROT.CTX ← %CTX.
%PROT.next ← the previous %PROT.
%STACK[%SI + 1] ← clause.ESC.
%STACK[%SI + 2] ← %CTX.
%STACK[%SI + 3] ← %PC + x.
%STACK[%SI + 4] ← %FUN.
%STACK[%SI + 5] ← %MOD.
%STACK[%SI + 6] ← %SP.
%SI ← %SI + 7.
%SP ← %SI.
%CTX ← clause § %CTX.

`(LEAVE-WATCH): 147`
%CTX ← %STACK[%SI - 1].
%ENV ← %STACK[%SI - 3].
%SI ← %SI - 3.
%PROT ← %PROT.next.

5.1.5 A word about bytecodes vs. primitives

Each previously described ULM primitive has been assigned specific bytecodes because they could not be implemented using virtual machine reified primitives. This means that just like the Scheme conditionnal is not reified, `ulm:thread`, `ulm:agent`, `ulm:when` and `ulm:watch` are not values but special forms.
The following ULM primitives have no conceptual reason to be assigned to bytecodes, which is why `ulm:signal`, `ulm:emit`, `ulm:await`, `ulm:pause`, `ulm:migrate-to` and `ulm:safe-migrate-to` are available as VM primitives (referred to by `NPREDEFINED`). But because they are frequently invoked rather than passed around as values, they have been given bytecodes too, which are described briefly here with no example.

5.1.6 Signal Creation

Signal creation with (`ulm:signal`) is very simple:

$\mathcal{C}[\![$(`ulm:signal`)$]\!]\rho$t =
    `(CREATE-SIGNAL)`

`(CREATE-SIGNAL)`: 143
    %VAL ← a new signal.

5.1.7 Signal Emission

Signal emission with (`ulm:emit` π) consists in evaluating π into a signal and emitting it:

$\mathcal{C}[\![$(`ulm:emit` π)$]\!]\rho$t =
    ($\mathcal{C}[\![\pi]\!]$ ρ false)§`(EMIT-SIGNAL)`

`(EMIT-SIGNAL)`: 144
    Emits the signal in %VAL.

5.1.8 Signal Awaiting

Signal awaiting with (`ulm:await` π) consists in evaluating π into a signal and awaiting it:

$\mathcal{C}[\![$(`ulm:await` π)$]\!]\rho$t =
    ($\mathcal{C}[\![\pi]\!]$ ρ false)§`(AWAIT-SIGNAL)`

`(AWAIT-SIGNAL)`: 156
    Awaits the signal in %VAL. If the signal is already emitted, do nothing, otherwise, yield to the scheduler.

5.1.9 Cooperation

Cooperation with (`ulm:pause`) simply cooperates:

$\mathcal{C}[\![$(`ulm:pause`)$]\!]\rho$t =
    (`PAUSE`)

`(PAUSE): 157`
    Cooperates until the next instant.

5.1.10 Migration

There are four forms of migration: unsafe subjective with (`ulm:migrate-to` $\pi_{\texttt{dest}}$), unsafe objective with (`ulm:migrate-to` $\pi_{\texttt{dest}}$ $\pi_{\texttt{agent}}$), safe subjective with (`ulm:safe-migrate-to` $\pi_{\texttt{dest}}$ $\pi_{\texttt{timeout}}$ $\pi_{\texttt{respawn}}$) and safe objective with (`ulm:safe-migrate-to` $\pi_{\texttt{dest}}$ $\pi_{\texttt{timeout}}$ $\pi_{\texttt{respawn}}$ $\pi_{\texttt{agent}}$). They are each assigned a different bytecode, but their compilation is straightforward: we push all but the last argument on the stack and the last argument in VAL:

$\mathcal{C}[\![$(`ulm:migrate-to` $\pi_{\texttt{dest}}$)$]\!]\rho$t =
    ($\mathcal{C}[\![\pi_{\text{dest}}]\!]$ ρ false)§(`MIGRATE-SELF`)

`(MIGRATE-SELF): 149`
    Marks the current agent for migration to the site in %VAL.

$\mathcal{C}[\![$(`ulm:migrate-to` $\pi_{\texttt{dest}}$ $\pi_{\texttt{agent}}$)$]\!]\rho$t =
    ($\mathcal{C}[\![\pi_{\text{dest}}]\!]$ ρ false)§(`PUSH-VALUE`)§($\mathcal{C}[\![\pi_{\text{agent}}]\!]$ ρ false)§(`MIGRATE-OBJ`)

`(MIGRATE-OBJ): 150`
    Marks the agent in %VAL for migration to the site in %STACK[%SI - 1].
    %SI ← %SI - 1.

$\mathcal{C}[\![$(`ulm:safe-migrate-to` $\pi_{\texttt{dest}}$ $\pi_{\texttt{timeout}}$ $\pi_{\texttt{respawn}}$)$]\!]\rho$t =
    ($\mathcal{C}[\![\pi_{\text{dest}}]\!]$ ρ false)§(`PUSH-VALUE`)§($\mathcal{C}[\![\pi_{\text{timeout}}]\!]$ ρ false)§(`PUSH-VALUE`)§
        ($\mathcal{C}[\![\pi_{\text{respawn}}]\!]$ ρ false)§(`SAFE-MIGRATE-SELF`)

`(SAFE-MIGRATE-SELF): 154`
    Marks the current agent for migration to the site in %STACK[%SI - 2] with a respawn order in %VAL, and a timeout in %STACK[%SI - 1].
    %SI ← %SI - 2.
    %VAL ← a new safe migration signal for the corresponding migration.

$\mathcal{C}[\![(\texttt{ulm:safe-migrate-to}\ \pi_{\texttt{dest}}\ \pi_{\texttt{timeout}}\ \pi_{\texttt{respawn}}\ \pi_{\texttt{agent}})]\!]\rho t =$
$(\mathcal{C}[\![\pi_{\text{dest}}]\!]\ \rho\ \text{false})\S\texttt{(PUSH-VALUE)}\S(\mathcal{C}[\![\pi_{\text{timeout}}]\!]\ \rho\ \text{false})\S\texttt{(PUSH-VALUE)}\S$
$(\mathcal{C}[\![\pi_{\text{respawn}}]\!]\ \rho\ \text{false})\S\texttt{(PUSH-VALUE)}\S(\mathcal{C}[\![\pi_{\text{agent}}]\!]\ \rho\ \text{false})\S\texttt{(SAFE-MIGRATE-OBJ)}$

`(SAFE-MIGRATE-OBJ): 155`
Marks the agent in %VAL for migration to the site in %STACK[%SI - 3] with a respawn order in %STACK[%SI - 1], and a timeout in %STACK[%SI - 2].
%SI ← %SI - 3.
%VAL ← a new safe migration signal for the corresponding migration.

5.1.11 References

Reference compilation is simple, but their execution in the VM uses a simple yet interesting trick: when attempting to access or modify a remote reference, we reset the PC to the current instruction before yielding to the scheduler. This means that the next time we will be scheduled the access or modify bytecode will be re-executed (and certainly succeed if we were rescheduled because the reference is now local) and the access or modification will then complete.
Aside from creating a reference, which cannot block, dereferencing and affecting a reference may block, and thus need to cooperate, which is why they cannot be implemented as VM primitives because VM primitives cannot cooperate freely for various reasons described in 6.4. Therefore they have been assigned bytecodes and do not exist as VM primitives. For consistency rather than implementation reasons, creating a reference has been assigned the same treatement.
Making a reference with (`ulm:ref` π) is simple:

$\mathcal{C}[\![(\texttt{ulm:ref}\ \pi)]\!]\rho t =$
$(\mathcal{C}[\![\pi]\!]\ \rho\ \text{false})\S\texttt{(MAKE-REF)}$

`(MAKE-REF): 151`
%VAL ← a new reference in the current thread's store to the data in %VAL.

Reference access with (`ulm:unref` π) is simple too, but since at runtime we reschedule the instruction in case of remote reference, we need to avoid using the VAL signal, which is not stored in the current thread but shared by the VM. So we save the reference on the stack.

$\mathcal{C}[\![(\texttt{ulm:unref}\ \pi)]\!]\rho t =$
$(\mathcal{C}[\![\pi]\!]\ \rho\ \text{false})\S\texttt{(PUSH-VALUE)}\S\texttt{(UNREF)}$

`(UNREF): 152`
Looks at the reference stored at %STACK[%SI - 1].
If it is local:

%VAL ← %STACK[%SI - 1].value.
%SI ← %SI - 1.
If it is remote:
%PC ← %PC - 1, then yields to the scheduler.

Reference modification with (`ulm:ref-set!` $\pi_{\texttt{ref}}$ $\pi_{\texttt{data}}$) uses the same pattern:

$$\mathcal{C}[\![(\texttt{ulm:ref-set!}\ \pi_{\texttt{ref}}\ \pi_{\texttt{data}})]\!]\rho t = (\mathcal{C}[\![\pi_{\text{ref}}]\!]\ \rho\ \text{false})\S(\texttt{PUSH-VALUE})\S(\mathcal{C}[\![\pi_{\text{data}}]\!]\ \rho\ \text{false})\S(\texttt{PUSH-VALUE})\S(\texttt{REF-SET!})$$

`(REF-SET!): 153`
Looks at the reference stored at %STACK[%SI - 2].
If it is local:
%STACK[%SI - 2].value ← %STACK[%SI - 1].
%SI ← %SI - 2.
If it is remote:
%PC ← %PC - 1, then yields to the scheduler.

5.2 Migration

In order to be able to send an agent from one site to another, several things need to be done. In this section we explain the transport mechanisms and serialisation process required for agent migration.

5.2.1 Transporting agents

Here we describe how and when agents are sent through the network.

Transport mechanisms

ULM agents are sent through the network between instants, which means they leave at the end of instant on one site, and arrive at the beginning of instant on the remote site. From a transport layer point of view, this means there is data to be sent at the end of instant, and to be received at the beginning of instant, through the network. But network transmissions are insecure operations which take an indefinite amount of time to complete.
While sending agent data through the network, there is no reason why the scheduler should be sending agents sequentially at the end of instant. Indeed, although the agents should leave at the end of instant, there is no reason why we should mandate that they should be received on the remote site before we start a new instant on the local site. Sending the agents through the network is an operation which does not have to take time at the end of instant, and delay the scheduling of a new instant. This is why all agents are sent on the network in threads asynchronous to the ULM scheduler: they are started at the end of instant, and can take all the time needed to perform their task before notifying the ULM scheduler of their success or failure.
Receiving agents from the network poses a similar problem: reading an entire agent though the network can take a long time and should not delay the beginning of the next instant. It also poses a new problem: if the ULM scheduler only starts reading agents at the end

of instants, this means incoming connections during the instant would have to wait until the end of instant to be accepted. This is a delay which may trigger timeouts on the sending end, and is not acceptable if we can do better. Additionally, it would be hard to decide the window of time during which the end of instant should accept incoming connections. To solve this, incoming network connections are accepted asynchronously to the ULM scheduler, so that agents are incorporated in the scheduler when they have been fully read from the network.
Notifications to the ULM scheduler of outgoing agents success or failure, as well as fully-read incoming agents, are buffered in a list which is checked by the ULM scheduler at the end of instant. This is how the scheduler can send safe-migration signals and incorporate immigrant agents.

Transport layers

We have two transport layers for ULM agents: TCP/IP and Bluetooth. Our use of TCP/IP to transport agents is relatively standard: we use the TCP port number 1027 (unassigned by IANA [IAN], the Internet Assigned Numbers Registry), although it is possible to specify an incoming port number for the ULM VM, and a destination port number for all agent migrations. ULM sites are referred to by host name or IP address with possibly a TCP port number for TCP/IP transport in calls to `ulm:migrate-to` and `ulm:safe-migrate-to`. TCP/IPv4 addresses are in the form `NNN.NNN.NNN.NNN[:port]` and TCP/IPv4 hostnames are in the form `fqdn[:port]` with the default port number used unless given.
The second transport layer is Bluetooth, more specifically the RFCOMM layer, which is a wireless serial connection. The RFCOMM layer has a concept of port number called *channels*, for a total of 30 channels. For such a small amount of channels available, using a fixed channel for ULM migration makes no sense, since every phone ships with several channels already assigned for different tasks. To overcome this, Bluetooth uses the Service Discovery Protocol (SDP) to associate *services* to channel numbers. When we want to start listening for incoming Bluetooth connections, we ask the system to provide us with a free channel. Then we connect to the local SDP server and associate the "ULM" service with the channel number we obtained. In order to know which channel to use when sending an agent to a remote Bluetooth device, we query the remote SDP server for the channel associated with the "ULM" service, and use it for the connection. Bluetooth addresses are in the form `XX:XX:XX:XX:XX:XX` with 12 hexadecimal numbers.

5.2.2 The use of modules for agents

Transporting an agent from one site to another requires not only a transport mechanism, but also a transport format. A migrating agent consists in several aspects we discussed earlier: its code, its continuation and its data. We already discussed the OULM file format, which specifies a way to encapsulate some data (the constants) and ULM code. The continuation consists in the thread stack, several runtime structures such as escapers, protectors and the context amongst others. The data we need to save consists in data types which can be created by ULM programs at runtime, including those which can exist as constants in ULM source code (pairs, numbers, strings, vectors...) and those we can only create at runtime: closures, primitives, mixin instances, references, signals, ports.
Representing a migrating agent also requires storing its bytecode, its list of global variables, closures, and all the constants it requires. We can see that the OULM file format is a

Name	Size	Contents
`ValuesCount`	2u	The number of serialised values
`Values`	? * n	The list of serialised values
`AgentIndex`	2p	The agent's index
`GlobalsCount`	2u	The number of GlobalValue fields
`GlobalValues`	2p * n	The indexes of each GlobalValue

Table 5.1: OULM `Agent` attribute

subset of what we need to represent migrating agents. We chose to create a module for each departing agent in which to store all its code, required functions and globals, and extend the OULM file format in order to add the missing information.
To that end we added an `Agent` attribute which specifies that the OULM file represents a serialised agent. It contains the fields described in Table 5.1.
All serialised non-constant value is stored in a format similar to the `Constant` fields, and their list is stored in the `Values` field. We call that list the *agent value pool*. The `AgentIndex` then points to the value representing the agent we are migrating. It is followed by a list of `GlobalValue` fields pointing to the values of each global variables defined in the agent's module.
In order to be able to differentiate references to constants from the constant pool and values from the agent value pool (runtime vectors can contain constant values), the agent value pool is indexed starting after the last constant. So if there are N constants in the constant pool, the first agent value index is N. All indexes pointing from 0 to N-1 refer to constants, while indexes greater than N refer to agent values. Note that since constants are not modifiable, they cannot contain pointers to agent values, so constants remain contained in the constant pool.
The agent value pool can contain all `Constant` types (although they will not contain `ImportedField` and `ExternField` types) plus the new types defined in Table 5.2.
We are now going to describe each new `ConstantType` in details.

Threads

Due to architecture and implementation reasons, our Bigloo VM has two layers for representing threads: a layer used to represent any kind of Scheme thread represented with the `VMState` constant type, and a layer representing the ULM-specific information regarding scheduling of such threads represented with the `Agent` and `Thread` constant types.
The `Thread` constant type represents a ULM thread with the fields described in Table 5.3. Note that the `Thread` constant type does not have a `Parent` field, this is because there is exactly one agent per agent module, and every thread within the agent module is a child of this agent. It is therefore easy to reassociate the agents and its children threads.
The `Name` field refers to a `Symbol` entry in the agent value pool. This name is not visible to the user but it is there in case future versions will find a use for it. The `VMState` field points to the thread's `VMState` entry in the agent value pool. The `WWCount` field indicates how many cells there are in the `WWContext` field to follow. The `WWContext` field is the thread's context, it contains an ordered list of `WhenClause` and `WatchClause` entries in the constant value pool, which represents the suspension and preemption context. The `LeafSignal` field

Type	ConstantType	Data size	Data
`Agent`	100	16 + ?	An agent
`Thread`	101	10 + ?	A thread
`VMState`	102	26 + ?	A thread's continuation and state
`WhenClause`	103	2	A suspension clause
`WatchClause`	104	4	A watch clause
`LocalRef`	105	4	A local reference
`RemoteRef`	106	4	A remote reference
`Closure`	107	6	A closure
`Env`	108	3 + ?	An environment
`Primitive`	109	2	A primitive
`Null`	110	0	A non-value
`Signal`	111	2	A signal
`SCMProtector`	112	16	A Scheme protector
`ULMProtector`	113	6	An ULM protector
`SCMEscaper`	114	2	A Scheme escaper
`ULMEscaper`	115	2	An ULM escaper
`InputPort`	116	0	An input port
`OutputPort`	117	0	An output port
`STDIN`	118	0	The STDIN input port
`STDOUT`	119	0	The STDOUT input port
`STDERR`	120	0	The STDERR input port
`Mixin`	121	2 + ?	A mixin instance

Table 5.2: OULM new constant types

Field	Size	Contents
`Name`	2p	Its name
`VMState`	2p	Its VMState index
`WWCount`	2u	The number of WhenClause/WatchClause to follow
`WWContext`	2p * n	A list of WhenClause/WatchClause
`LeafSignal`	2p	A leaf signal or null
`PreemptClause`	2p	A WatchClause or null

Table 5.3: OULM `Thread` constant type

Field	Size	Contents
`Name`	2p	Its name
`VMState`	2p	Its VMState index
`WWCount`	2u	The number of WhenClause/WatchClause to follow
`WWContext`	2p * n	A list of WhenClause/WatchClause
`LeafSignal`	2p	A leaf signal or null
`PreemptClause`	2p	A WatchClause or null
`RefCounter`	2p	A reference counter
`RefSignal`	2p	The signal used for its references
`MigrationSignal`	2p	A safe migration signal or null

Table 5.4: OULM `Agent` constant type

points to either a `Signal` entry or a `Null` entry in the agent value pool. If it is a signal, it is the leaf signal the thread is waiting for using `ulm:await`, or a signal used internally to represent waiting on a remote reference. If it is null, there is no leaf signal to wait for. The `PreemptClause` field points to either a `WatchClause` (which should also be in the `WWContext` field), or a `Null` entry. It points to a `WatchClause` if that preemption cell was previously activated (on the departure site) and should be unwound to. It points to `Null` if the thread has no pending preemption.
The `Agent` constant type has the same fields plus some additional ones at the end. It is described in Table 5.4.
The common fields have already been discussed. The `RefCounter` field points to a counter in the constant pool representing the number of references created by the agent. This is used in order to assign unique names to references and could be implemented differently. The `RefSignal` is the signal used internally to represent waiting on a remote reference: each agent has such a signal, which is emitted at each instant for each present agent. If a thread needs to wait on a remote reference, it will be waiting for the reference's agent's signal as a leaf signal. The `MigrationSignal` points to either a `Signal` entry or a `Null` entry. It will be non-null if this agent requested a safe migration, in which case it will point to the signal that has to be emitted upon arrival in order to indicate migration success.
The `VMState` entry contains the low-level entries needed by the VM to execute threads. It is described in Table 5.5.
The `PC` and `SP` fields contain respectively the thread's program counter and stack pointer. The `StackCount` field contains the number of stack entries to follow, while the `Stack` field contains a list of stack entries. Note that it contains only meaningful entries in the stack: those between zero and the StackIndex register, not all the blank space available for the stack to grow. This space can be allocated by the arrival site. The other fields are exactly as described earlier in the implementation of the VM, except for the `Specifics` field, which contains a mapping between names and values and represents the *thread-specific storage*.

WWContext

The `WhenClause` entry represents a suspension clause and contains only one field representing the suspension signal. It is described in Table 5.6.
The `WatchClause` entry represents a preemption clause and contains the preemption signal

Field	Size	Contents
`PC`	2u	The Program Counter
`SP`	2u	The Stack Pointer
`StackCount`	2u	The size of the stack
`Stack`	2p * n	The stack
`Env`	2p	Its environment
`Fun`	2p	The current function
`Module`	2p	The current module name
`Protector`	2p	The current protector or null
`Escaper`	2p	The current escaper or null
`EscaperRet`	2p	Return value for the escaper or null
`InputPort`	2p	The current input port
`OutputPort`	2p	The current output port
`ErrorPort`	2p	The current error port
`SpecificsCount`	2u	The number of specifics
`Specifics`	(2p + 2p) * n	The thread-specific data

Table 5.5: OULM `VMState` constant type

Field	Size	Contents
`Signal`	2p	The suspension signal

Table 5.6: OULM `WhenClause` constant type

Field	Size	Contents
`Signal`	2p	The preemption signal
`Escaper`	2p	The escaper

Table 5.7: OULM `WatchClause` constant type

Field	Size	Contents
`Value`	2p	The reference's value
`Name`	2p	Its unique name

Table 5.8: OULM `LocalRef` constant type

and its associated escaper. It is described in Table 5.7.

References

References can be either local or remote to the agent, and are represented respectively by the `LocalRef` and `RemoteRef` entries. References use a unique `Name` field internally to identify them. The `LocalRef` constant type contains the data of the reference. It is described in Table 5.8.
The `RemoteRef` constant type on the other hand does not contain the reference's value but a signal which will be emitted when the remote reference becomes local. This is the signal present in the agent which created the reference (or the site). It is described in Table 5.9.

Functions

There are two types of functions reified to the user: closures and primitives. Closures are ULM functions represented by the `Closure` constant type. It is described in Table 5.10.
The `FunDescIndex` field contains the index of the function descriptor representing the closure within its module. The `Module` field points to its module name, which can be either the current agent module or an ubiquitous module. The `Env` field points to an `Env` entry which describes the linked list of environments as such. The `Env` constant type is described in Table 5.11.
The `NextEnv` points either to the next environment as an `Env` entry or to a `Null` entry if there is no next environment. The `Values` field contains the list of values in this environment.
Primitives are represented by the `Primitive` constant type, which only contains their name since primitives are ubiquitous values. The `Primitive` constant type is described in Table

Field	Size	Contents
`Signal`	2p	The reference's signal
`Name`	2p	Its unique name

Table 5.9: OULM `RemoteRef` constant type

Field	Size	Contents
`FunDescIndex`	2u	Its FunctionDescriptor index
`Env`	2p	Its captured environment
`Module`	2p	Its module name

Table 5.10: OULM `Closure` constant type

Field	Size	Contents
`NextEnv`	2p	The next environment or null
`ValuesCount`	1u	The number of argument values
`Values`	2p * n	The list of argument values

Table 5.11: OULM `Env` constant type

5.12.

Null

Fields which can have a null value are represented using the `Null` constant type, which contains no field.

Signals

Signals are represented using the `Signal` constant type, which contains a unique name field. It is described in Table 5.13.

Protectors

Protectors are the data structures used to represent things to be done while unwinding the stack in case of preemption. They come in two flavours: Scheme protectors represented by the `SCMProtector` constant type, which is introduced by `unwind-protect` and the ULM protectors represented by the `ULMProtector` constant type, which are in charge of restoring the preemption/suspension context as we unwind the stack.
The `SCMProtector` constant type contains the following fields, which have been described in the previous chapter. It is described in Table 5.14.
The `ULMProtector` constant type contains only the stack index, next protector and the suspension/preemption clause which should be restored at the top of the context. It is described in Table 5.15.

Field	Size	Contents
`Name`	2p	The primitive's name

Table 5.12: OULM `Primitive` constant type

Field	Size	Contents
`Name`	2p	The signal's unique name

Table 5.13: OULM `Signal` constant type

Field	Size	Contents
`StackIndex`	2u	The stack index
`NextProtector`	2p	The next protector or null
`SP`	2u	The Stack Pointer
`PC`	2u	The Program Counter
`Env`	2p	The environment
`Module`	2p	The module
`Fun`	2p	The FUN register
`Ret`	2p	The protector's return value

Table 5.14: OULM `SCMProtector` constant type

Field	Size	Contents
`StackIndex`	2u	The stack index
`NextProtector`	2p	The next protector or null
`WWClause`	2p	The WhenClause/WatchClause to restore

Table 5.15: OULM `ULMProtector` constant type

Field	Size	Contents
`StackIndex`	2u	The stack index

Table 5.16: OULM `ULMEscaper and SCMEscaper` constant types

Field	Size	Contents
`FieldCount`	2u	The field count
`Fields`	(2p + 2p) * n	The fields

Table 5.17: OULM `Mixin` constant type

Escapers

Escapers are the data structure representing a target endpoint for unwinding. The `SCMEscaper` represents the type of function given to `bind-exit` for escaping its body, while the `ULMEscaper` is used internally to represent the endpoint of a weak preemption introduced with `ulm:watch`. Both types of escapers contain only one field representing the stack index where the rest of the escaping information is stored. Both constant types are described in Table 5.16.

Ports

Serialising a port correctly can be problematic. There are several types of input and output ports in Scheme, such as console, file, network, procedure, or string ports among others. A procedure port or a string port could be serialised easily, but console, file and network ports refer to underlying I/O structures provided by the Operating System, which are most often not serialisable. For example: a given file may or may not exist on different sites, a network socket is not transportable since there is no support for relocating socket endpoints in most network protocols. While we could transport the file along with the file port, or leave a socket proxy for socket ports, these are not necessarily good solutions, and we have not yet studied this problem. Until (if) we find a suitable solution, file and network ports are serialised to closed ports represented by the `InputPort` and `OutputPort` constant types which contain no fields.
At this time we do not support procedure and string ports, so they have no been assigned constant types. We have several ubiquitous values to represent the standard input, output and error ports as given by the system. These are console ports that cannot be serialised, although it makes sense to map them to ubiquitous values so that printing to standard output or reading from the standard input always works. They are represented using the `STDIN`, `STDOUT` and `STDERR` constant types with no field.

Mixins

Mixins instances are represented by a list of field name/value associations using the `Mixin` constant type. It contains the fields described in Table 5.17.
The `Fields` field is a list of 2p `Name` and 2p `Value` fields pointing to a `Symbol` entry for the field's name and its value.

5.2.3 What happens to data

In order to pack everything that the agent needs into an agent module and in the OULM format, we need to figure out exactly what data the agent needs. This is done by the process of serialisation. During serialisation, we need to find all the data required by the agent in order to complete its execution. We do this by traversing the agent's *root*[1] which is the structure we use to represent the agent in the VM.
Indeed the agent contains its state, its stack, from which we can find all the functions required by its continuation and all values pushed on the stack. From the functions which do not come from ubiquitous modules we can traverse their bytecode in order to find objects which will be accessed by this bytecode, such as constants, global and imported references, functions and predefined variables. From those functions we also find the captured environments we need to serialise.
All the constant values we traverse, and all the constant references we find in the bytecode we traverse are assigned a new constant index. We also assign constant indexes for imported and extern references we find in the bytecode and those used by the OULM format (attribute names, global names and module name).
All the non-constant values we traverse are also assigned a number used for the agent value pool. This includes strings, symbols, keywords, pairs, vectors, the agent, its children threads, its preemption and suspension clauses, the closures, environments, primitives, signals, protectors, escapers, ports and mixin instances.
The ULM references are also assigned such a number, but they require special care. When an agent leaves with a reference it owns, the reference will be serialised as a `LocalRef` constant type, and the system will mark the reference object as being remote from now on since the agent has left with it. Other threads which had a pointer to such references will now see it as remote. Departure of remote references are not affected since it will stay remote for both the agent and the departed site. Upon arrival, references local to an agent should be merged with any remote reference representing the same reference which were on the arrival site prior to the agent's arrival. In the same way, any remote reference brought by the agent which were pointing to a store local to the arrival site will be merged with their local counterparts. In both cases the previously remote references are transformed to local references. Arriving remote references pointing to stores not present on the arrival site are left remote.
A related treatment is required for ubiquitous values such as signals, primitives and standard ports. Ports representing the standard input, output and error are detected and serialised as such. When arriving on a new site, these ubiquitous values are deserialised not as new values but as their local counterpart. Primitives and standard ports should always have a local counterpart, but for signals if they do not they are deserialised to a new signal value (as long as the signal's unique name is kept it will remain ubiquitous).

5.2.4 What happens to the bytecode

For every non-ubiquitous closure we traverse from the agent's root, we need to absorb its bytecode and function descriptor in the agent module we are building. We call this process *phagocyting*. Closures which belong to ubiquitous modules are not phagocyted, they are simply serialised, but their bytecode will be available on the remote site, and as such it is not traversed either. For every phagocyted closure we need to add its function descriptor

[1] This concept of root is similar to that of garbage collectors.

and bytecode to our agent's module, after which we traverse the bytecode and modify it if needed.
As we traverse the bytecode of functions we traverse, we come across several bytecodes which indicate that there is something new we should traverse and possibly serialise. For example, when we come across bytecodes referencing global variables. We already discussed that global variables are cloned and absorbed: this is where it is done. For each bytecode accessing a global variable, we phagocyte this global variable by adding it to the agent's module. We give it a new index and replace the bytecode so that it will properly reference the phagocyted global. We also serialise its value, which is stored in the `Agent` attribute.
Imported reference bytecodes pointing to ubiquitous modules are left alone, but those pointing to non-ubiquitous modules are phagocyted as global variables: they are assigned a new global variable slot, and the bytecode is changed from `IMPORTED-REF` to `GLOBAL-REF`. Its value is also serialised.
Bytecodes such as `CREATE-CLOSURE`, `CREATE-THREAD` and `CREATE-AGENT` also trigger the phagocyting of the function descriptor they refer to and the traversal of its bytecode.
Then there are various implementation artifacts which need to be corrected during bytecode phagocytation: at runtime we apply some Just In Time modifications to some bytecodes in order to speed up their execution when we reexecute them. This includes replacing costly `IMPORTED-REF` bytecodes which reference a variable using its name and module's name by a `CHECKED-IMPORTED-REF` bytecode which uses the module's index in the list of loaded modules and the index of its global. A similar trick is applied to primitives which are referenced by name but can be sped up using their local index. These and several other local modifications to the bytecode are undone during phagocytation so that the serialised bytecode remains portable.
Once everything has been traversed, serialised and phagocyted, the agent is packed in the OULM format and sent to the remote site using TCP/IP or Bluetooth where it is unpacked, unserialised and integrated to the scheduler.
Although serialisation and deserialisation would be nice to do asychronously as we do for sending and reading on the network, we do not think it is easily doable nor productive enough. Indeed we have seen that certain values have to be modified by serialisation and deserialisation, and these modifications should not happen after the EOI when an agent leaves or before the EOI when an agent comes. For example, references should not change state (local/remote) within an instant. So serialisation and deserialisation is still done sequentially at the EOI.

5.3 Scheduling

The scheduler we have shown in the Semantics section of the Language chapter is complicated. Yet the semantics of the language are simple, though subtle, but the truth is that they could be explained in just a few paragraphs of text. Very broadly we could say that each thread is scheduled until every thread is waiting for the next instant or an absent signal, at which point we do the end of instant phase, *rinse and repeat*.
This seems simple enough, yet there are several aspects of this type of scheduling which are very inefficient: every thread needs to be considered by the scheduler several times during an instant, even when it has already been determined as waiting. Signals have to be put in a set every time they are emitted, then looked up in order to know whether they have been emitted. The end of instant has to traverse every thread possibly several times in order to

prepare the next instant.
The mere fact that we are walking every thread several times at the end of instant introduces a delay between instants proportional to the number of threads. As this delay grows, it becomes noticeable by the programmers, who may be tempted to pack more and more things to do within an instant, in order to change instants less often. In the case of many threads doing graphical work, such as cellular automatas (a typical example of the use of FairThreads [Bou04c]) where there are as many threads as cells (250 000 threads in our benchmarks), the end of instant becomes visually noticeable and thus has the unpleasant property of being annoying.
We have found several techniques to speed up scheduling, by following the idea that scheduling time should not be proportional to the number of threads, but to the number of things to care about while scheduling. This includes not spending time on threads we know cannot be scheduled, putting the load of waking up threads on signal emission instead of between each micro-instant, and scheduling preemption as soon as we can be certain it would happen at the end of instant. With the idea in mind that it is the emission of signals which marks certain threads for scheduling or triggers preemption, we have spread the actions previously done at the end of action or between micro-instants within the instants: during signal emission which should be an action taking a time proportional to the effect the signal emission will have.
These techniques have been experimented first in a C implementation of ULM's primitives without migration, called LURC (Lightweight ULM/Reactive in C). LURC is a C library with support for several thread backends and ULM's threads, signals, suspension, preemption and protection all with the same semantics as ULM without migration (and therefore references). LURC is presented in detail in the Annex chapter of this dissertation because its scheduling techniques have been ported to ULM's scheduler, and it presents interesting and unique aspects of thread implementation in C.
In this section we present the various implementation techniques we developed to spread the scheduling cost during the instant and speed up scheduling wherever possible. It should be noted that these optimisations could be used for a more traditional scheduling of the end of instant, with little difference in speed or efficiency. Our goal was not necessarily to produce the fastest ULM scheduler but the fastest ULM scheduler with the minimal time spent in the EOI, thus increasing the chance that the time spent in the EOI would be comparable to the time spent during thread cooperation within the instant.
As far as the scheduling semantics is involved, it should be noted that our scheduling implementation greatly resembles in practise the semantics described earlier, but differs in several parts. We have chosen to describe the traditional semantics earlier because they are much more easy to grasp, but the scheduling is very inefficient in the light of several techniques described in this section. We accept the slight difference in semantics because in most cases the scheduling is similar, and when it differs, the actual scheduling is the same for every execution of the same program, and follows an intuitive scheduling (for example: the first thread to wait for a signal is the first to be notified).

5.3.1 The End of Action phase

The EOI (End Of Instant) phase has already been discussed several times: it is the phase between instants where the scheduler has to do several things. This includes sending and receiving migrating agents, triggering preemptions, pruning terminated threads, figuring

out which threads to schedule for the next instant and resetting every signal to the non-emitted status.
In the hope of spreading the load incurred by this phase throughout the instant, we had the intuition that there are several scheduling operations done at the EOI which should be possible to do earlier. This is based on the simple observation that there are obvious cases where a thread's scheduling at the next instant is obvious in several cases. For example a thread consisting of:

```
(ulm:thread
 (let loop ()
  (ulm:pause)
  (loop)))
```

This is a flagrant case of unusefulness from the part of this thread, which does nothing but skip instants using `ulm:pause`. Based on the semantics and the facts that this thread has no suspension or preemption contexts, that this thread does not use signals or strong preemption, and that this is an ininite loop, it is obvious that this thread **will** be scheduled at the next instant. And we know this right from the moment it cooperates with `ulm:pause`. Why should we spend time on this thread at the EOI when we know perfectly well what its next instant's scheduling will be *as soon as it cooperates*?
In fact cooperation with `ulm:pause` is special because it introduces the absolute certainty that the thread will not be scheduled again during the current instant, and as such we know the thread will not call any more ULM primitives until the end of instant. In fact, except for eventual signals which may trigger its preemption at the EOI, its state *cannot* change from its cooperation point until the EOI. For all these reasons, we call the point when a thread cooperates with `ulm:pause` the thread's *End Of Action* phase: a phase when we can plan its next instant's scheduling in most cases.
Take the following example of a thread reaching its EOA phase where we know what will happen at the EOI with certainty:

```
(define kill (ulm:signal))
; emit it to trigger the preemption
(ulm:emit kill)
; execute this body for at most one instant
(ulm:watch kill
 ; this is executed
 (print "one")
 ; we trigger the EOA
 (ulm:pause)
 ; this is never reached
 (print "two"))
; this is printed instead of "two"
(print "done")
```

The above example uses the `ulm:now` pattern in order to execute an expression for at most one instant through the use of a *surefire* preemption: a preemption which we are certain will be triggered. When the thread reaches its EOA, we know it is in a preemption block with an emitted signal. Since the thread will not change state until the EOI, and signal cannot be un-emitted, we know the preemption will take place at the EOI, and we also

know that since there is no other preemption, suspension or protection context that the thread will be scheduled at the next instant and will resume at the point where it will print "done". This is a scheduling step which can be done as early as the EOA and does not necessitate any waste of time at the EOI.

A similar observation can be made of threads in suspension blocks:

```
(define kiss (ulm:signal))
; emit it to get in the suspension
(ulm:emit kiss)
; execute this body unblocked for at most one instant
(ulm:when kiss
 ; this is executed
 (print "one")
 ; we trigger the EOA
 (ulm:pause)
 ; this is never reached
 (print "two"))
```

In this example, once the thread reaches its EOA it will enter an infinite sleep until it is kissed[2] (that is, when the `kiss` signal is emitted). Although the scheduler may not know that the signal is never going to be emitted, we know at the EOA that the thread will be scheduled at the next instant only when the `kiss` signal is emitted. If there is a queue of threads to execute when the signal is emitted in a future instant, that's where we should put this thread, and not look at it until that signal is emitted in a future instant. It is interesting to note that since the thread reached its EOA in a suspension context, it means the `kiss` signal was emitted, and therefore the queue we previously talked about, which may have contained threads waiting for it, must has been cleared when the signal was emitted earlier at this instant. We will talk more about that in 5.3.2.

There are many places in which the next instant's scheduling of a thread can be determined with certainty at its EOA. This is why we promoted `ulm:pause` to a primitive from this derived form:

```
(define (ulm:pause)
 ; execute this body for at most one instant
 (ulm:now
  (lambda ()
   ; wait forever on a new signal which will
   ; never be emited
   (ulm:await (ulm:signal)))))
```

Since `ulm:now` is based on `ulm:watch` and `ulm:await` on `ulm:when`, it is hard for the scheduler to deduce the fact that the suspension signal will never be emitted and that the thread will not be rescheduled during this instant. This is made explicit by promoting `ulm:pause` to a primitive.

[2]Possibly by a prince on a white horse, but this is not relevant, nor is the fact that it does not seem to be going to happen.

5.3.2 Wait queues and their recycling

If we want to have an efficient scheduling, and most synchronisation is based on signals, we need efficient signals. The first thing we have to decide is how to efficiently determine the status of a signal. The semantics we described earlier marks emitted signals by storing them in a set of emitted signals. Although this set can be implemented by a hash table, which has a fair lookup performance, it requires memory allocation for each emitted signal and deallocation at each EOI to reset the signals.

A better technique has already been used for reactive signals for a while, in FairThreads variations such as Junior and Sugarcubes [Sus01]. It consists in attributing a number to each instant, and storing the current instant number in the signal during emission. This way marking a signal as emitted consists in writing a number, and questioning its status consists in comparing the signal's number with the current instant's. Switching to the next instant consists in incrementing the current instant number, which has the side-effect that each previously emitted signal becomes automatically non-emitted since their signal number is out of date[3].

FairThreads and its variations also used a system of *wait queues*, a system by which we store the list of threads waiting for a signal in the signal itself. When emitting a signal with `ulm:emit` it is then possible to examine this list of waiting threads and put it in the list of threads to schedule later in the current instant. In the case of ULM however, things are not so simple: a thread with several suspension clauses may be waiting for several signals at once before it can be rescheduled. In order to wait on several signals at once it follows that the thread should be located in each signal's wait queue, and only be scheduled once each signal has been emitted. This turns out to be complex: if we remove the thread of each wait queue one the signal has been emitted, this means we have to put it back in each wait queue at every instant. Now suppose the thread has all but one of its suspension signals emitted, yet the last signal is not emitted before the EOI. This means it will be in one signal's wait queue at the EOI and has to be placed back in every other wait queue, which means the scheduler spends time on an unscheduled thread.

Let us imagine another strategy: if we want to avoid any operation at the EOI to re-register threads in their suspension signal's wait queues, we should leave them in the wait queues when each signal is emitted. But then determining whether a thread can run after a signal emission requires examination of each of its suspension clauses to see if they have all been emitted. It is hard to speed up using a counter for the number of signals the thread is waiting for, because we would need to reset that number at each EOI. On the other hand, if the thread is waiting for N signals, the cost of checking every remaining signal at each signal emission is quadratic.

Because every suspension signal needs to be emitted in order to schedule a thread, we don't need to put that thread in every signal's wait queue: only one suffices. Indeed, suppose we register the thread only in the signal which will be emitted last (of all the thread's suspension signals): when the last signal is emitted we notice that all the signals have been

[3]The observant reader will notice that computer fixed-width integers loop if incremented too many times. We have calculated that a system running today on a 64bit system using 64bit integers for instant numbers with only one thread doing nothing but skipping instants will run for 25 years before the instant number loops. We find this acceptable considering the machine has to run continuously on the same hardware for 25 years, which is very extreme. Of course this fails if we are able to upgrade the machine's hardware without shutting it down. If at all possible, we can use 128bit integers but one current 64bit hardware the performance gain/cost may need to be reassessed.

emitted and can reschedule the thread. If on the other hand the thread is registered in one of the signals which is not the last to be emitted, we can leave that signal's wait queue and register in one of the suspension signals not yet emitted. In the worst case we are quadratic again, but in the best case we are linear.
The fact that a thread is registered in only one of its suspension signal also means we have nothing to do at the EOI to re-register threads in wait queues: if the thread reaches the EOI while waiting on a signal, it will have to wait for that signal at the next instant just as well. If on the other hand the thread reaches the EOI and is not waiting on a signal, it means the thread must have been scheduled, and therefore reached its EOA phase. During its EOA we can plan its next instant's scheduling because we know what to do: if it has no suspension clauses, it will be running at the next instant, if it has it will be waiting for any one of its suspension signals[4].
Let us suppose a thread with at least one suspension signal which reaches its EOA. Note that reaching the EOA implies the thread has been scheduled. If it has been scheduled and has a suspension context, it means the thread is not in any wait queue anymore, since we clear the wait queues when we emit signals. Because signals cannot be emitted twice during an instant, it follows that each emitted signal has empty wait queues. This means that we can put anything we want in these wait queues, they will not be examined until we reach the next instant by incrementing the instant number, thereby resetting every signal to non-emitted. This is a clear opportunity to register threads in their wait queues for the next instant. This is how we recycle wait queues: prior to a signal's emission it contains threads waiting for the signal at the current instant, and after the signal's emission it contains threads waiting for that signal starting at the next instant. This is how we plan future suspension for threads reaching their EOA.

5.3.3 Fast lanes for simple waits

We have shown how we can use wait queues for multiple signals suspension. The power of nesting suspension clauses requires each wait queue to be traversed upon signal emission, and for each thread further examination is required in order to find a potential next signal to wait for. Yet in many of our programs we noticed that we were only waiting on leaf signals using `ulm:await`. Since this primitive cannot be nested, it is much simpler than `ulm:when`, and when a thread is waiting on a leaf signal with no suspension clause above it we can do a much faster scheduling, since we know for sure that when the signal is emitted the threads waiting for it will be scheduled (since they are only waiting on one signal).
To that end, threads waiting for a signal using `ulm:await` with no suspension clause are put in a special queue in that signal, called the *simple wait queue*. Upon signal emission, that queue is taken directly out of the signal and appended to the list of threads to schedule in constant time. This guarantees a constant time for both waiting and emitting such simple signal uses.

5.3.4 Planning weak preemption

Weak preemption and its associated protection clauses are much harder to plan ahead of the EOI. There are some simple cases, several complicated cases and a few impossible cases

[4]This blatantly ignores the complexity that weak preemption and protection can introduce here, but they are discussed below.

which we will present here. In many cases, planning the preemption is only half the work: figuring out what will have to be done about the preemption is a whole lot of fun.

Surefire preemption at the EOA

The simplest case of planning weak preemption (ahead of the EOI) is when a thread reaches its EOA with a surefire toplevel preemption. This is the case when a thread calls `(ulm:now (lambda () (ulm:pause)))` with no other preemption or suspension context. In this function we emit a signal, then enter a weak preemption block where we call (eventually) `ulm:pause` and reach the EOA. At this point we know the preemption will be triggered, and the thread will be scheduled at the next instant, with the task of unwinding to the `ulm:watch` call. We can then mark the preemption in the thread, and put it in the list of threads to schedule at the next instant.

Marking preemption when emitting

Now let us take the same example, but with an unsure preemption:

```
(define kill (ulm:signal))
; enter an unsure preemption
(ulm:watch kill
 (ulm:pause))
```

At this thread's EOA it is harder to plan ahead because we don't know if the `kill` signal will be emitted later. At the next instant, the thread may or may not be preempted, although it will be scheduled in both cases. We will put the thread in the list of threads to schedule at the next instant, but we cannot mark the preemption as a sure thing. So we use yet another queue in the signal: the *preemption queue* where each thread is registered when it enters a preemption block on that signal.
Upon a signal emission, its preemption queue is traversed and each thread is checked. We have to remember that if a thread has several triggered weak preemptions only the outermost weak preemption is triggered. Indeed it does not make sense to trigger an inner preemption block if an outer one is triggered too. To that end each preemption clause in the thread's context is assigned an increasing number, with the outermost clause having zero and inner clauses greater than zero. So we only mark preemptions with the lowest number, in order to mark only outer preemptions.

Planning weak preemptions for waiting threads

The remaining cases of weak preemption planning before the EOI are more complex and involve a lot of special cases which are too obscure to delve into here, so we will only outline them.
Whenever we emit a signal we traverse the list of threads this signal preempts. When such a thread has already reached its EOA we have seen it is simple, but if the thread is to be scheduled later, or if the thread is waiting, things get more complex. In fact if the thread is to be scheduled later it can only relinquish the control by reaching its EOA (case we have already described), or waiting for an absent signal. So the complex case is always how to preempt a thread waiting for a signal before the EOI.

There are various techniques for planning a preemption on a waiting thread, and they depend on various aspects of the thread waiting. Although we will not discuss our techniques in detail here because they are slightly complex, we can successfully plan weak preemption for waiting threads in several cases. For some cases though we have to resort to putting the preempted thread in a list which will be examined at the EOI to take the proper action.

Conditional preemption and caching

We have now seen every case of activated preemption, but there are cases of unsure preemptions: when we emit a preemption signal for suspended preemption clauses like this:

```
(define kill (ulm:signal))
; preempt at the next instant
(ulm:thread
 (ulm:pause)
 (ulm:emit kill))

(define wake (ulm:signal))
; enter the suspension context
(ulm:when wake
 ; enter the preemption
 (ulm:watch kill
  (ulm:pause)))
```

In this example we emit the preemption signal when the preemption block is suspended because of `wake`'s absence at the second instant. This preemption is not taken into account unless the `wake` signal is emitted later during the second instant. To that end we have a *suspended preemption* field in the thread, where we mark any suspended preemption which may be of interest later. Of course we only mark suspended preemptions if there are no outer surefire preemption already marked. When we traverse the suspension signal's wait queue because of its emission later, we look at possible preemptions and promote them to planned preemptions if their outer suspension clauses are all satisfied.
Many lookups, such as preemption marking, finding outer or inner suspension clauses, or the number of satisfied suspension clauses are sped up by using clause numbers and caches which are invalidated at each instant with the same technique as signals are reset: using instant numbers.
Such aggressive planning of the next instant's scheduling and preemption, as well as the many uses of cache requires more memory per thread, although no memory is allocated during the scheduling. We have reduced the number of such optimisations for the J2ME VM to the cost of spending a little bit more time at the EOI for more cases of preemption, because J2ME devices really come with low memory and it is more desirable to schedule more threads slower than crash as soon as we create too many faster threads[5].

5.3.5 Minimal End Of Instant

So far we have succeeded in planning each thread's next instant's scheduling before the EOI except for the degenerate cases where we have to switch a thread's wait queue as a result

[5]Ironically we would only cause the crash to occur faster by having a faster scheduling.

of preemption. The end of instant is reduced to treating the degenerated cases, serialising outgoing agents and starting their emission on the network asychronously, deserialising incoming agents and schedule them, then increment the instant number to automatically reset every signal and cache, replace the list of threads to schedule now with the list of threads to schedule at the next instant, then schedule the first thread.

Chapter 6

Native interface (s)

+++ Divide By Cucumber Error. Please Reinstall Universe And Reboot +++

– Terry Pratchett

No, darling ! I'm sure they drive on the left over here in France.

– Famous last words

In many computer languages, there are ways to invoke code which belongs to a lower level language, such as invoking assembly code from C, or C code from Java or Scheme. There are several uses for such lower level code: sometimes a lower level language can be optimised in ways the higher level compiler could not achieve, sometimes it is necessary to operate on hardware, or interact with the OS, and sometimes it is just used to wrap a useful library written in a different language. The means to access a lower level language are often called the *native interface* because the lower level language often happens to be the implementation, or native language of the higher level one.
We have developed a native interface in ULM, which we present here because it has some interesting properties.

6.1 Syntax

Since we have several virtual machines for ULM, the lower level language to ULM can be different between VMs. For instance it makes sense to make the native interface's language similar to that of the VM itself, because it has to communicate to the VM in order to create and manipulate ULM types. So for the Bigloo VM the native interface would be Bigloo Scheme, while the J2ME VM would target the Java language.
In order to call a native function from ULM, it has to be imported using the module's `extern` clause. This clause contains a list of native modules and for each of these modules, the list and prototypes of their exported functions, plus *backend*-specific information (that is, specific to the type of ULM VM used). Let us look at the following example:

```
(module fou
 (extern (barre ((bigloo . barre) (java . "org.acme.barre"))
          (f x y)
          (g f L))))

(g (lambda (x) (f x 2)) '(1 2 3))
```

This program shows a module `fou` importing a native module `barre` (the french spelling equivalents of foo and bar[1]). This native module exports the `f` and `g` functions with the prototypes `(f x y)` and `(g f L)`, and supports two backends: Bigloo and Java. The Bigloo backend will provide the `barre` module in a `barre` library, while the Java backend will provide it in the `org.acme.barre` Java class. How these two backends provide native modules for ULM, and how they can be created are described in the following two sections.
It is not necessary to give prototypes for native functions in an extern module clause, the name of the imported variable is enough, but it is desirable to give prototypes in order to have compiler warnings in case of misuse. It would be nice if we did not have to specify the list of imported variables from a native module, but then we would have to either extract that list from each (or at least one) backend, or implement a sort of header file for each native module describing which variables it exports to ULM.
Extern variables imported from extern modules are then treated in the same way as variables imported from other ULM modules: they are read-only.

6.2 Bigloo backend modules

Native modules for the Bigloo backend are written in Bigloo Scheme. For our last example, here is how we would write the Bigloo module `barre`:

```
(module barre
 ; this is required for dynamic loading
 (eval (export-exports))
 (export (f x y)
         (g f L)))

(define (f x y)
 (+ (* 2 x) y))

(define (g f L)
 (map (lambda (x)
       ; we cannot invoke f as a Bigloo procedure:
       ; it is a ULM type, so we use the invocation
       ; API from the ULM VM
       (ulm-vm:invoke f x))
      L))
```

This is a simple module exporting `f` and `g`. If we look into details, we notice that the Bigloo VM uses the Bigloo numbers and lists to represent ULM numbers and lists, otherwise `+`, `*`

[1] Which we assure you has nothing to do with calling crazy a past (and now passed on) prime minister.

and `map` would not work on the values we get as arguments from the ULM VM. But because the ULM VM does not represent ULM functions with Bigloo functions, they have to be invoked via a special API exposed by the ULM VM. We could automagically wrap (or box) ULM functions given to the Bigloo native layer, but this would be at the cost of traversing every object passed as parameter to and possibly returned from native invocations, so we prefer to make the call explicit. But let us not look into such sordid details anymore.

This Bigloo module is then compiled by Bigloo into ... well actually that's a good question: what is it compiled to? Bigloo also supports several backends: C, Java and .NET. Depending on which backend is used, the compilation will produce different files: object files for C, class files for Java and portable object files for .NET. Since this code has to be loaded in the Bigloo ULM VM later, it should be compiled to the same backend as the Bigloo ULM VM. Furthermore it has to be packaged in a format which is loadable dynamically by the VM, since we load modules just-in-time. The typical standard packaging for dynamic libraries in C, Java and .NET are respectively a shared-object library (SO lib on most systems), a Jar file and a DLL file. The compiled Bigloo module has to be packaged in the appropriate format for the Bigloo ULM VM.

When the VM triggers an extern module load with the `IMPORTED-REF` bytecode for an `ExternField`, such a package is looked up by the VM (in our example `libbarre.so`, `barre.jar` or `barre.DLL`). Then it is loaded by the VM which uses the backend mappings to find the correct Bigloo module within the archive. In our example, the Bigloo VM would load the `barre` Bigloo module.

The value of the referenced extern variable is then read by the Bigloo VM from the loaded module, and automatically boxed as a ULM primitive value if the variable is a procedure. This way, in ULM invoking that extern procedure is transparent (although it is less transparent in the extern Bigloo procedure itself). On the other hand, if the Bigloo variable contains a data structure containing Bigloo functions or any type of value not used in ULM, those will not be boxed, so there is still room for improvement later.

6.3 Java backend modules

For the Java ULM VM, things are a bit different, because the underlying backend language is more different from ULM than Bigloo Scheme. In order to define a ULM module and its variables it takes a bit more effort in the Java VM:

```
 1: // remember our module comes from org.acme.Barre
 2: package org.acme;
 3:
 4: // import the ULM VM classes
 5: import ulm.vm.*;
 6:
 7: // extend the NativeModule type
 8: public class Barre extends NativeModule {
 9:
10:   // declare a number for each procedure
11:   public final static int F = 1;
12:   public final static int G = 2;
13:
14:   // now instantiate a primitive for each procedure. the arguments are:
15:   // ULM name, number, arity, variable-arity, and owner module to
16:   // export the procedure
17:   public final Primitive f = new Primitive("f", F, 2, false, this);
18:   public final Primitive g = new Primitive("g", G, 2, false, this);
19:
20:   // the constructor calls the super constructor with the "barre"
21:   // symbol. this is the ULM name of this module
22:   public Barre(){
23:     super(Symbol.getSymbol("barre"));
24:   }
25:
26:   // Overload the invocation function for procedures of arity 2
27:   public Object invoke(VirtualMachine vm, int index,
28:                        Object p1, Object p2){
29:     switch(index){
30:     case F:
31:      int x = ULM.ULM_TO_INT(p1);
32:      int y = ULM.ULM_TO_INT(p2);
33:      return ULM.INT_TO_ULM(2*x + y);
34:     case G:
35:      Cons L = ULM.ULM_TO_CONS(p2);
36:      Cons ret = Cons.nil;
37:      while(L != Cons.nil){
38:        ret = new Cons(vm.invoke(p1, L.car), ret);
39:        L = (Cons)L.cdr;
40:      }
41:      return ret.reverse();
42:     default:
43:       return super.invoke(vm, index, p1, p2);
44:     }
45:   }
46: }
```

In order to define a native module, one has to create a subclass of `ulm.vm.NativeModule`,

whose constructor expects the ULM name of the module (line *23*). The principle is the same as discussed earlier for declaring ULM primitives in the J2ME VM: a primitive is an instance of the `ulm.vm.Primitive` class with a pointer to its module, an index, a name and an arity. Each primitive instance must have a different index in a given module. These indexes are defined as global constants in lines *11* to *12*. We then create all the instances of these primitives and store them in global variables (in case we need to access them directly in this module) in lines *17* to *18*. The arguments given to the primitive constructor are: its ULM name, index, arity, boolean for variable arity and owner module. This constructor is in charge of registering the primitive in the current module, so they can be looked up by the VM when accessed by name in the bytecode.
When such a primitive is invoked it is dispatched to its owner module, which then invokes the `invoke` method corresponding to the primitive's arity (there is one `invoke` methods per given arity), where the primitive's code is dispatched depending on the primitive index. In our example, we have two primitives of arity 2. The code for the `f` native function is at lines *30* to *33* and consists in unboxing the two number arguments to Java integers, then performing some operation and boxing the result which is then returned.
The code for the `g` native function is at lines *34* to *41* and consists in unboxing the `L` (given here as `p2`) argument to a pair (implemented with the `ulm.vm.Cons` class), and walking this list pair by pair. While we traverse the list, we use the VM's `invoke` method to invoke the ULM function given as the `f` argument (here `p1`) on each element. All these results are kept in a list which we then return once we have reversed it in order to preserve the original list's order.
The default case of the switch at line *43*, which invokes the super-method is here only to throw an error if the implementer declared a primitive but forgot its code, or placed it in the wrong `invoke` function.
This class is then loaded an instantiated by the VM when the `barre` native module needs to be loaded. Since there is no `ClassLoader` in J2ME this class needs to be compiled and placed in the same JAR as the VM on the mobile platform: it cannot be packaged in its own JAR as is the case for the Bigloo VM with Java backend. Because J2ME does not support reflection it is only possible to invoke a class' constructor if it has no argument when loading classes dynamically, as is the case for our native modules. This is why the `VirtualMachine` instance used by the native module has to be given to each `invoke` method instead of being given to the module's constructor and stored for each invocation.

6.4 Reentry

In the previous examples, we have seen how native function are invoked from ULM, and we have seen in the native function how to invoke ULM functions from the native code. But this invocation of ULM code from native code hasn't been explained. We have talked in great length in a previous section why we use a virtual machine with no state stored in its continuation in order to implement migration. We have to go back to that explanation to understand why invoking ULM code from native code can be a problem in our VM.
When executing ULM code, the bytecode is devised precisely so that the VM will never require to store information required by ULM threads in the VM's continuation. This means that the part of the VM where it iterates over bytecode and schedules threads is built as a loop with no recursion: the VM's stack never grows between one bytecode execution to another. This is essential in order to be able to migrate threads and has already been

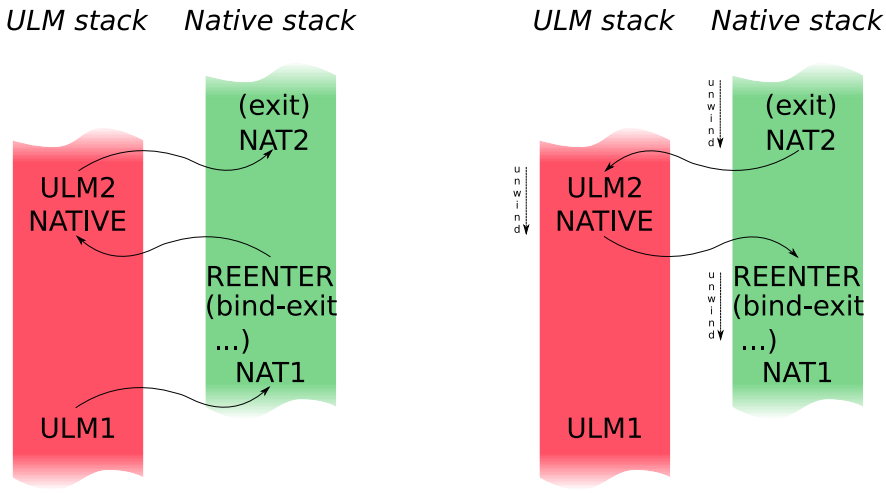
explained in the Implementation: Scheme chapter. It is also important because the VM does not have to rely on native threads to store state for each ULM thread: they can all be executed in the VM's thread whose stack does not grow.

When the VM invokes a primitive, its stack does grow because it is using its native language's invocation mechanisms and we need the primitive to return to the bytecode loop in order to return the return value of the primitive to the ULM program. The VM's primitives are devised so that it is guaranteed that each primitive returns to the VM without invoking any ULM function. But it is desirable for user native modules to be able to invoke ULM code from within their native functions. This is what we do in our `g` native function: we invoke the `f` ULM function gotten as argument. In order to be able to invoke this ULM function, the VM has to execute its bytecode: it has to enter the bytecode loop. Then it has to return the function's return value to the native function. Because the native function wants to use the native language's method invocation mechanism, this requires the use of the native continuation: the stack in our case. When the VM invokes a native function, its stack grows, if this function then invokes our VM's bytecode loop again, the stack grows again, and we have the bytecode loop in two call frames in the stack. This can be repeated if the ULM function invokes another native method. Note that this is not the case for the VM primitives where we take great care not to invoke ULM code and return from the primitive without growing the VM's stack.

Now suppose the ULM function `foo` invokes the native method `g`, which in turn invokes the ULM function `bar`. We have two VM bytecode loop call frames in the VM's stack. Suppose now that the `bar` ULM function cooperates with another thread, which is then installed as the current thread in the latest VM bytecode loop. It is now clear that this second thread has call frames in the VM stack which belong to the other thread. Not only can we not migrate this thread (it is not unreasonable to forbid threads with native call frames to migrate), but if this thread returns from its current function, it will return to the native function `g` which it never invoked because it was invoked by the other thread. This is not reasonable at all.

We call the process of invoking VM bytecode from native code *reentry*: a thread leaves the VM to invoke native code, then reenters the VM if that native code invokes ULM code. The process of reentry is not a problem with no migration and if each ULM thread was executed in its own native thread: the VM could then easily have several bytecode loop call frames in its stack without the possibility of mixing up threads. But using a native thread per ULM thread presupposes a mapping throughout the entire lifetime of each thread: if is not possible to map a ULM thread to a native thread only when it invokes a native function. We believe this is a waste of resources, since we would be wasting the resources of native threads for every ULM thread which does not use reentry, which is more an exception than the norm.

We have implemented a mechanism to fix this: we delegate native calls in another stack. The VM never calls reentering native functions in its own stack, but delegates the call to another thread. This other thread can then be created only when needed by the VM. It will wait for the VM to signal a native call to be done, at which point it will invoke the native function and the VM will wait for its return value. If at one point the native function wants to reenter the VM, the auxiliary thread will signal the VM that it needs to invoke ULM code. The VM will then place an appropriate call frame in the ULM thread's stack and invoke the appropriate reentry function without growing its stack. When the reentry is done, the auxiliary thread is notified by the VM which gives it the function's return value,



(a) Doing the reentry

(b) Unwinding from reentry

Figure 6.1: Reentry

at which point it returns to the native call, which can then return to the VM by signaling the VM thread and passing it the native function's return value.
With this system it is possible to have several native calls and reentries per ULM thread: in that case the same auxiliary thread's stack is reused by the same ULM thread. Since the VM never has any stack growth, and each ULM thread's native calls are located in distinct native threads, cooperation is no longer a problem. When an auxiliary thread returns to the VM with an empty stack (no pending reentries left) it can be destroyed or kept around to be recycled for another ULM thread when needed.
Upon migration, the scheduler checks each migrated ULM stack for the presence of reentry stack frames, and aborts the migration of the agent if one of its thread (or itself) has pending native calls.

6.4.1 Unifying protectors and exception handlers

Let us suppose an `ulm1` ULM function making a native call to `nat1`, which in turn reenters into the `ulm2` ULM function which invokes the `nat2` native function. The call stack is thus `ulm1 > nat1 > ulm2 > nat2`. Now let us suppose that `nat2` wants to unwind to `nat1`. This can be done in a language-dependant manner: by throwing an exception in Java, or via `call/cc` or `bind-exit` in Bigloo.
The resulting ULM and native stacks are shows in Figure 6.1 a. This figure represents the stacks growing upwards, with a Bigloo native backend, where we use `bind-exit` in `nat1` to unwind in `nat2`. The NATIVE mark in the ULM stack is the special stack frame to indicate the VM must return to the thread's native stack.
If we let this native unwinding be done without handling, we will allow unwinding ULM's `ulm2` function without executing the protection blocks. Of course this is not what we want. What we want is shown in Figure 6.1 b: we want any unwinding from the native stack to unwind *through* the ULM stack and call the appropriate protectors. The techniques to achieve this are very backend-dependent.

```
; function which intercepts any unwinding from f
(define (intercept f . args)
 ; use this flag to detect unwinding
 (let ((unwinding-detected #t))
  (unwind-protect
   ; compute (f args)
   (let ((ret (apply f args)))
    ; if we arrive here there was no unwinding
    (set! unwinding-detected #f)
    ; return a flag saying it is a normal return
    (cons 'return ret))
   ; this is the protector
   (if unwinding-detected
    ; if we detected unwinding, return a flag
    ; marking it
    (cons 'unwind '?)))))
```

Figure 6.2: Detection of unwinding

In the Java backend, the only way of unwinding is through exception throwing. When we execute a native call from the VM, we install a *catch-all* exception handler to capture and prevent any exception from unwinding. When such an exception is captured, the native call returns to the VM, which then calls the protection blocks in ULM (in `ulm2`) until the next native call frame (in `nat1`), where it resumes the throwing of that exception. If the exception is not captured in the last native call frame (in `nat1`), it is transformed into an ULM error and thrown as a non-continuable exception in `ulm1`.

Although we could choose to represent Java exceptions as ULM exceptions, which would be thrown from native code to ULM code, it would break ULM's system of exceptions, where handlers are supposed to be called without any stack unwinding, whereas in Java exception handlers only work with stack unwinding. Thus in order to reach the handler in `nat1` we would have to unwind past the `ulm2` ULM function, which is not the expected behaviour.

In Bigloo, catching the unwinding is more complex: even if we forget `call/cc` (since we do not support it in ULM it makes no sense to support it for our Bigloo native backend), we have to handle calls to `bind-exit` made to unwind the stack. This means that when we call a native Bigloo function, we have to prevent it from unwinding. Such unwinding can be detected using `unwind-protect` as shown in Figure 6.2.

We use `unwind-protect` with a boolean variable in order to determine whether a function call unwound the stack or returned normally. But detecting it is not enough: we need to stop the unwinding from progressing. We can do this by ignoring the protection block by exiting it with `bind-exit` as shown in Figure 6.3.

But this is still not enough: we need to be able to resume the unwinding if `ulm2` unwinds completely, and we reach the next native call frame `nat1`. Before aborting the protector's continuation (with `bind-exit` in our last example), we need to store that protector's continuation so it can be used later. For instance when we are done unwinding from `ulm2` we need to start unwinding again in `nat1`. There is no alternative to `call/cc` this time, as shown in Figure 6.4.

```
; function which intercepts any unwinding from f
(define (intercept f . args)
 ; calling exit will leave this function
 (bind-exit (exit)
  ; use this flag to detect unwinding
  (let ((unwinding-detected #t))
   (unwind-protect
    ; compute (f args)
    (let ((ret (apply f args)))
     ; if we arrive here there was no unwinding
     (set! unwinding-detected #f)
     ; return a flag saying it is a normal return
     (cons 'return ret))
    ; this is the protector
    (if unwinding-detected
     ; if we detected unwinding, return a flag
     ; marking it, by ignoring the continuation
     (exit (cons 'unwind '?)))))))
```

Figure 6.3: Preventing the unwinding

```
; function which intercepts any unwinding from f
(define (intercept f . args)
 ; calling exit will leave this function
 (bind-exit (exit)
  ; use this flag to detect unwinding
  (let ((unwinding-detected #t))
   (unwind-protect
    ; compute (f args)
    (let ((ret (apply f args)))
     ; if we arrive here there was no unwinding
     (set! unwinding-detected #f)
     ; return a flag saying it is a normal return
     (cons 'return ret))
    ; this is the protector
    (if unwinding-detected
     ; if we detected unwinding, capture the
     ; unwinding continuation
     (call/cc (lambda (cont)
                ; return a flag marking the unwinding,
                ; by ignoring the continuation and saving
                ; it for later
                (exit (cons 'unwind cont)))))))))
```

Figure 6.4: Detecting and delaying the unwinding

This way when we call a native function from ULM, we know whether it returned normally, or if it tried to unwind. If it tried to unwind, we unwind until the next native call frame `nat1` where we resume the unwinding by calling the captured continuation. In theory, the last native call frame cannot unwind further, since there is no further native call frame which could have installed the unwinder.
For Bigloo exceptions handlers (those from SRFI-34), we have found no way to interleave ULM exception handlers with Bigloo's handlers. We can detect exceptions which are thrown, but we cannot detect whether they will unwind, whether they are *continuable* (it is a property of the exception throwing, not the exception), nor can we access the current list of handlers in order to interleave exception ULM and Bigloo handlers.
For all these reasons we chose to not interleave Java or Bigloo exceptions with ULM code, but represent them as unwinders instead and invoke protection blocks. If the native code wants to throw ULM exceptions it is possible, but native exceptions are not wrapped automatically.
As for the other way around: code which unwinds from ULM through native code, we unwind through Java call frames by throwing an `UnwindingException`, and from Bigloo call frames by using a more elaborate version of the unwinder we install with `bind-exit` as previously described.

Chapter 7

Reactive Event Loop

> *I always avoid prophesying beforehand, because it is a much better policy to prophesy after the event has already taken place.*
>
> – Winston Churchill

There are many similarities between ULM's model of cooperative threads using signals for synchronisation, and event loops. In fact, we can see event loops as a form of cooperative scheduling. In this chapter we will describe the similarities and differences between event loops and ULM's model. We will explain why it is necessary to integrate some event loop features in ULM, and how this is done. We will then proceed to show why we need ULM's scheduler to interact with various event loops, and how this can be done.

7.1 Event loops and ULM

7.1.1 Presentation of the event loop

An event loop is a form of cooperative scheduling without threads: it consists of a loop which emits certain *events* when certain conditions are met. There are various types of such event-triggering conditions but the most common ones are input/output (IO) notifications, timeouts and idle notifications (an idle event is emitted when there are no other events to emit).
Programs can register callback functions to events. These functions will then be called by the event loop when their events are emitted. These functions have to be *atomic*, that is, they have to complete and return to the event loop before any other action can be taken.
We can think of the event loop as the ULM scheduler, events as signals, and callbacks as threads executing a function when a signal is emitted. The main difference is that the callback functions are atomic whereas with ULM threads, we would not need to return to the event loop, we could simply cooperate to it and resume execution where we left it at the next instant.
Event loops and ULM are very similar, except that we think we can do more with ULM threads than with atomic callback functions.

7.1.2 Why we need event loops

Event loops are commonly found in many languages whether they support *real* threads (not mere event loop callbacks) or not. They are found in client/server programs which are

driven by input/output processing, and many graphical user interface (GUI) libraries.

Client/server example

Client/server programs use event loops in order to handle simultaneous blocking IO calls without using threads. Suppose a program wants to handle incoming connections on two sockets at the same time. It is not desirable to poll the sockets repeatedly in loop until one of them shows an incoming connection, this is commonly accepted as a waste of computer resources. In most systems there is a way to suspend the execution of a program until a certain IO condition is met. In POSIX for example, the `accept` system call suspends the execution of the program until there is an incoming connection in the specified socket. This is one traditional example of *blocking* IO: a call to an IO function which blocks the caller until completion of the IO operation. Most IO functions such as reading, writing, creating files, sockets, pipes are blocking operations by default.
If we want to accept incoming connections on any of two sockets at the same time we have to use another means. In ULM's world of cooperative threads we would have another problem with the same cause: any call to a blocking IO operation would block the current thread but not cooperate, thus every other thread would be blocked too, which is clearly not what we want. This is where asynchronous threads show their use: we could have two asynchronous threads, each blocked while waiting on one socket without blocking the other thread.
Using asynchronous threads to handle blocking IO calls is indeed the approach chosen by Bigloo's FairThreads library [SBS04]: it is possible to *detach* a cooperative thread from the scheduler and make it asynchronous while it completes the blocking call without blocking other cooperative threads. But what if we do not want the cost of asynchronous threads, and their dangerous integration in the middle of deterministic cooperative threads. When we want a deterministic execution this is the equivalent of having a wolf in sheep's clothing[1].
Back to our client/server example with two sockets, if we want to listen to two sockets at once without asynchronous threads we have to use *non-blocking* IO calls. There are two traditional ways of doing non-blocking calls. The first consists in informing the system we want to perform a certain IO operation, which should not block, and the system will notify the program later of the operation's successful or unsuccessful completion. The other way consists in waiting for the *readiness* of IO structures (sockets, files, pipes) before calling an otherwise blocking IO call which will not block because it was ready for that call.
Doing non-blocking IO calls with delayed notification of completion is complex because after each IO call the program cannot be sure of its success or failure, furthermore, the delivery of completion status by the system is often asynchronous and can thus lead to various related problems. Testing the readiness prior to IO calls is therefore the preferred choice in many cases. This can be done with `select` in POSIX, which usually resides at the core of any event loop.
To get back to our example again, in order to watch two sockets at once and respond to any incoming connection, we would enter a loop where we call `select` by asking it to return when there is any incoming connection on either of the two sockets. When this happens we would accept the connection without blocking (since we know there is an incoming connection waiting) and reply, then close the connection and loop.
This is precisely how event loops work: if we were to write our client/server program with

[1] Or to use a more actual analogy: a rottweiler play with children.

an event loop, we would register the two sockets on the event loop, and associate any incoming connection event to a callback function which would print the date. In ULM we think a similar solution could be used in order to avoid the need for asynchronous threads for blocking IO operations: if there was a way to register an IO event with the scheduler, a signal could be emitted by the scheduler to notify of the event occurrence (incoming connection for instance).
We also think that ULM's use of threads would solve a common problem with event loops. Suppose that instead of replying to incoming connections with the date, we would need some sort of *handshake* between the client and the server. Each would have to write and read from the socket several times in order to complete the handshake. Since these operations can potentially block, in an event loop, we would register callback functions for incoming/outgoing readiness. Since a callback function is atomic, it would not be able to save state easily, and we would have to split the handshake sequence in many callbacks, one for each step of the handshake. This is unpleasant work at best, one which would be avoided if using cooperating threads which could just wait for the next emission of the event signal at every step without blocking the other threads.

GUI example

The use of event loops in GUIs is slightly different than for client/server programs. In most GUIs event loops are used to centralise operations which need to be done on the graphics hardware, input devices and the logical components of the GUI. For example, the GUI's event loop would have a handler for mouse or keyboard activities, for the window manager's incoming requests to redraw parts of the GUI which have been hidden by another window, and to queue any request for redrawing, or inserting new elements in the GUI.
Traditionally, GUIs do not require threads, which is why they usually function with event loops. This conclusion [Ous96] was highly debated at the time when many new GUI libraries appeared, but is nowadays widely accepted by the majority of GUI libraries in use. Suppose a text editor window, used to write documents, with a blinking cursor. The GUI's event loop would have a *timer* event registered which would trigger the cursor's colour to change (white, black, white...) every second. Such a change of colour would then queue an *idle* event which would trigger a redraw of the cursor to the graphics hardware when the event queue has time for it. In the meanwhile there is an IO handler registered on the keyboard which would trigger the insertion of a character, thus queueing another redraw request. At the same time, the mouse is watched and external redraw requests by the system are honoured.
GUIs usually do not require threads, but they do not forbid the program which uses them to use threads. Some event-loop-based GUIs allow threads but forbid any thread outside the event loop from doing any GUI operation (SDL for example) while others (GTK or Java's Swing) take the appropriate precautions from asynchronous threads, and queue most work resulting from other threads' calls in the event queue for later synchronous (and safe) completion.
Anyone willing to write a GUI in ULM would thus surely need in addition to the IO signals we described earlier, the timer and idle types of signals to be registered to and emitted by the scheduler. Additionally, it may be an interesting research direction to design a GUI which allows callbacks to be more complex than atomic callbacks (like threads), while still retaining the safety of synchronous operations.

Type	ConstantType	Data size	Data
`IOSignal`	121	5	An IO signal
`TimeoutSignal`	122	12	A timeout signal
`IdleSignal`	123	2	An idle signal

Table 7.1: OULM new constant types

Field	Size	Contents
`Name`	2p	Its name
`Port`	2p	Its port index
`Events`	1u	The events to watch

Table 7.2: OULM `IOSignal` constant type

7.2 The new signals

In order to be able to write efficient client/server programs or GUIs without the need for asynchronous threads, we have decided to integrate several features of event loops into ULM in what we call the *Reactive Event Loop* (REL).
To that end, we have devised three types of signals which will be spontaneously emitted by the scheduler at the beginning of instants (like safe migration signals) when certain conditions are met. The *IO* signal will be emitted when an IO port becomes ready. The *timeout* signal will be emitted when a certain timeout has elapsed. The *idle* signal will be emitted when there are no other threads to schedule.
Each new type of signal is in fact a subtype of the normal ULM signal, which means they can also be emitted by any thread (although that may not make much sense), waited for and treated just like a plain signal. The only differences lies in the fact that they may be spontaneously emitted by the scheduler, and that they are serialised using new constant types described in Table 7.1.

7.2.1 The IO signal

A new IO signal is created with `(rel:io-signal port events)` which takes a Scheme port and a bitfield specifying the events of interest on the given port. The `events` argument can be a disjunction of `rel:port-read` for input events, `rel:port-write` for output events and/or `rel:port-exc` for exceptional events (some types of sockets support this).
IO signals are serialised into the `IOSignal` constant type described in Table 7.2.

7.2.2 The timeout signal

A new timeout signal is created with `(rel:timeout-signal secs usecs respawn)` which takes a number of seconds and micro-seconds, as well as a boolean respawn argument. The signal will be emitted after the specified interval has elapsed, and if the `respawn` argument is true, it will continue to be emitted after every interval.

Field	Size	Contents
`Name`	2p	Its name
`Seconds`	2p	Seconds index
`uSeconds`	2p	uSeconds index
`StartSeconds`	2p	Start seconds index
`StartuSeconds`	2p	Start useconds index
`Respawn`	2p	Respawn index

Table 7.3: OULM `TimeoutSignal` constant type

Field	Size	Contents
`Name`	2p	Its name

Table 7.4: OULM `IdleSignal` constant type

Timeout signals are serialised into the `TimeoutSignal` constant type described in Table 7.3.
Note that we need to store the time at which we started to count the timeout interval. This time is stored in seconds and microseconds relative to the beginning of the first of january 1970 in GMT so that it can be portable across sites.

7.2.3 The idle signal

A new idle signal is created with `(rel:idle-signal)`. This signal will be emitted at the beginning of instants where no thread is ready to be scheduled. Contrary to most other types of signals, it makes sense to have only one idle signal per site, with a name common across all sites, so that it will effectively be a ubiquitous value. Thus there is only one idle signal returned by `rel:idle-signal`, shared by all threads across all sites.
Idle signals are serialised into the `IdleSignal` constant type described in Table 7.4.

7.3 Example

In order to demonstrate how more natural it is to program using the REL than with traditional event loops, we are going to look at a simple typical client/server program. This is the *echo* server: a server that waits for incoming connections on a server socket, and when an incoming connection arrives, the server will read all the data from the client, then write all the data back before closing the connection.

7.3.1 Event Loop

Our first version of the *echo* server will be written using a traditional event-loop. Callbacks for IO activity can be registered using the `(el:add-io-listener port type callback data)` function. The given `callback` function will be called by the event loop whenever the given `port` is available for reading or writing, as selected by the given `type` argument which can be `el:port-read` or `el:port-write`. The callback function will be called with

two arguments: the port on which there is activity, and the given `data` argument, which is really the only place where we can store the state needed by the callback. The callback function should return `#t` to stay registered in the event loop, or `#f` if it should be removed from the event loop.

Once our main function `start-server` has created a server socket, and registered a callback for incoming connections on this socket, it enters the event loop by calling `(el:enter-loop)`, which will wait for activity for all registered listeners until every listener is removed.

Our `incoming-connection` callback will accept the incoming connection and register an `incoming-message` callback in order to read all the data from the client socket. Because this data can come in several burst, the `incoming-message` callback will read all the data available without blocking, and wait until activated again by the event loop for the remaining data. The data we accumulate between each activation from the event loop is kept in the `data` argument. This is how we store state between callback calls in event loops, which we find not very intuitive.

When all incoming data has been read, we register a callback to be called when the socket is available for writing, and keep this callback registered until all the data has been written.

Here is the full example:

```
; this is our main entry point.
(define (start-server)
 ; create a server socket listening on port 1234
 (let ((socket (server-socket-open 1234)))
  ; in order to be notified on incoming connections
  ; we register a callback on read events
  (el:add-io-listener socket el:port-read
                      ; the callback function:
                      incoming-connection
                      ; the callback data
                      #f))
 ; do the event loop
 (el:enter-loop))

; this is our callback when there is an incoming connection
; on the server socket which is given as first argument.
; the second argument is not used.
(define (incoming-connection ssocket data)
 ; accept the incoming connection
 (let ((client-socket (server-socket-accept ssocket)))
  ; register a listener for the incoming message
  (el:add-io-listener client-socket el:port-read
                      ; the callback:
                      incoming-message
                      ; the callback data
                      '())
  ; return true so that the event loop keeps this callback activated
  ; for future incoming connections
  #t))
```

```
; this is called when there is incoming data on the client socket.
; the data contains a pair whose car is the accumulated data read.
(define (incoming-message socket data)
 ; do a non-blocking read, so we read only what's available
 (let ((str (read-string-nb socket)))
  ; check if we have reached the end of the message
  (if (eof-object? str)
   (begin
    ; we are done reading, let's write it back when we can
    ; by adding a "write" listener which will be called every
    ; time we can write on the socket without blocking
    (el:add-io-listener socket el:port-write
                               ; the callback:
                               outgoing-message
                               ; the callback data:
                               data)
    ; return false so that this incoming-message listener is
    ; removed from the event queue
    #f)
   (begin
    ; add str to the data already read
    (set-car! data (string-append (car data) str))
    ; return true so that the event loop keeps incoming-message
    ; registered for further read events
    #t))))

; this is called when we can write on the client socket without
; blocking. the data contains a pair whose car is the remaining
; data to be written.
(define (outgoing-message socket data)
 ; do a non-blocking write, attempting to write all the data.
 ; it returns the number of characters successfully written
 ; until the socket would block.
 (let ((n (write-string-nb socket (car data))))
  ; see if we wrote everything
  (if (= n (string-length (car data)))
   (begin
    ; we are done writing everything, remove this callback from
    ; the event loop by returning false after we close the socket
    (socket-close socket)
    #f)
   (begin
    ; we wrote a part of the data, let's remove that part
    (set-car! data (substring (car data) n
                              (string-length (car data))))
    ; return true to instruct the event loop to keep this
    ; callback registered for further write availability
    #t))))
```

The main thing we can see from this previous example is that the three loops (incoming connection, incoming message, outgoing message) are scattered across several functions and recursion cannot be used inside the loops (although admittedly we instruct the event loop about future recursion with the callback's return value).

7.3.2 Reactive Event Loop

Using the REL, we are able to write the *echo* server program using more traditional loops. In ULM we would write the `start-server` function as a loop which starts a new thread for each incoming connection. This thread would then loop over the incoming data, then loop again over the outgoing data. The main difference with asynchronous threads here is that each loop is in a suspension block on a REL signal. For instance, in the `start-server` loop, this REL signal will be emitted every time there is data to be read on the server socket, which means there is an incoming connection to be accepted. Whenever our suspension block is not suspended, we can accept a connection.

Here is the full example:

```
; this is our main entry point.
(define (start-server)
 ; create a server socket listening on port 1234
 (let* ((socket (server-socket-open 1234))
        ; in order to be notified on incoming connections
        ; we register a signal on read events
        (ssig (rel:io-signal socket rel:port-read)))
  ; enter a suspension block on this signal
  (ulm:when ssig
   ; loop over incoming connections.
   ; since we are in a suspension block, the fact that we are
   ; executing the next line means that the signal has been
   ; emitted and that there is an incoming connection, so the
   ; following line will not block
   (let loop ((client-socket (server-socket-accept socket)))
    ; create a thread to handle the client connection
    (ulm:thread
     (handle-connection client-socket))
    ; cooperate to reset the signal
    (ulm:pause)
    ; and loop for the next connection
    (loop (server-socket-accept socket))))))
```

```
; called in a thread for each client
(define (handle-connection socket)
 ; create one signal for incoming data, and one for outgoing data
 (let* ((rsig (rel:io-signal socket rel:port-read))
        (wsig (rel:io-signal socket rel:port-write))
        ; read all the incoming data in a suspended block on the
        ; read signal
        (data (ulm:when rsig
                (let loop ((data ))
                 ; do a non-blocking read, so we read only
                 ; what is available
                 (let ((str (read-string-nb socket)))
                  ; check if we have reached the end of the message
                  (if (eof-object? str)
                   ; return the data
                   data
                   (begin
                    ; cooperate to reset the signal
                    (ulm:pause)
                    (loop (string-append data str)))))))))
  ; now write the data back
  (ulm:when wsig
   (let loop ((data data))
    ; do a non-blocking write, attempting to write all the data.
    ; it returns the number of characters successfully written
    ; until the socket would block.
    (let ((n (write-string-nb socket data)))
     ; see if we still have to write
     (when (< n (string-length data))
      ; cooperate to reset the signal
      (ulm:pause)
      ; remove the data already written
      (loop (substring data n (string-length data)))))))))
```

We find that with the REL, we are able to write event-loop programs in a style much closer to asynchronous thread programming, without the need for asynchronous threads. This means that we are able to benefit from a more intuitive programming style while not having to pay the price of asynchronous threads.

7.4 Implementation

REL signals are emitted at the beginning of instant, but each type of REL signal requires a different method to determine emission. We will describe how we determine the set of signals which have to be watched, emitted, and how the REL integrates with the end of instant.

7.4.1 Which REL signals are watched?

Since ULM is a Scheme variant, it uses a garbage collector for its memory model, which means that rel signals are not expected to be *freed* or *terminated* by the programmer. This means that there may be many REL signals which are not used by the program anymore (those not referenced anymore) on which the REL should not spend time.

Because the ULM VMs have to keep a mapping of signals, agents, references and modules by name, and because of the unavailability of weak references on J2ME, there is a custom GC procedure in the ULM VMs for signals (among others). We use this GC to determine when the REL should stop watching the REL signals.

There is an optional optimisation by which the REL only watches REL signals which are currently awaited by at least one thread. This means that the REL signals which are still referenced but which no thread is currently waiting on, are not considered by the REL. This has the implication that REL signals may take one extra instant to be delivered. Indeed look at the following program:

```
; make an IO signal for a socket input
(define rel-sig (rel:io-signal socket rel:port-read))

(let loop ()
 ; wait for incoming data
 (ulm:await rel-sig)
 ; non-blocking read
 (let ((s (read-string-nb socket)))
  (unless (eof-object? s)
   (print s)
   ; reset the rel signal
   (ulm:pause)
   (loop))))
```

At the first instant, we await the IO signal, at the EOI the signal is emitted (we suppose there is some data), at the second instant the thread reads and prints the data, then pauses to reset the signal. At the third instant the IO signal is not considered by the REL because nobody is waiting for it. Then the thread returns from `ulm:pause` and loops, starts awaiting the IO signal, rinse and repeat.

It is important to understand why, if the REL only considers awaited signals, this IO signal cannot be emitted at the third instant: because nobody is waiting for that IO signal at the beginning of the third instant. This can cause some delay as the only thing our thread will be able to do during the third instant is to await the signal, which can not be emitted within the third instant because the REL is only considered during the EOI.

Had the REL considered every signal (not just those awaited), the third instant would have begun with the IO signal (possibly) emitted by the REL and the thread would not have had to wait for the fourth instant to read the data which was available at the third. Since this may or not matter depending on the program, this optimisation is optional at run-time. Note that the following variant of our example is faster in both cases:

```
; make an IO signal for a socket input
(define rel-sig (rel:io-signal socket rel:port-read))

; only work when there is incoming data
(ulm:when rel-sig
 (let loop ()
  ; non-blocking read
  (let ((s (read-string-nb socket)))
   (unless (eof-object? s)
    (print s)
    ; reset the rel signal
    (ulm:pause)
    (loop)))))
```

Semantically it is the same code, but it is faster for two reasons. First the IO signal is always marked in use as long as we do not exit the suspension block, so the REL always considers it. Second, but this is a by-product of our previously-described End Of Action optimisations, the thread is not scheduled between the return from `ulm:pause` and `ulm:await` because it starts waiting for the `rel-signal` as soon as it calls `ulm:pause`.

7.4.2 The IO signals

The IO signals are treated differently depending on the type of port being watched. String or procedure ports (currently not supported) are treated as if they are always ready for input or output. File or socket ports are watched by watching their underlying native file descriptors.
The implementation of how we watch native file descriptors differs on the various backends we use: we use the POSIX layer for the Bigloo VM's native backend, the NIO library of Java for the Bigloo VM's Java backend and our J2SE VM, and there is no support for the J2ME VM. There is no support for files in J2ME, and the NIO library is not available on that platform, so there is no way to ask for the readiness of several IO channels on J2ME, therefore IO signals on file and socket ports on the J2ME VM are not supported.
For the POSIX layer we use the native file descriptors and `select` to watch them. `select` takes a set of file descriptors to watch for input, output and exceptions, and a timeout (more on the timeout later). When we want to test the readiness of our file and socket ports, we call `select` with their underlying file descriptors and based on `select`'s return values, we emit the associated IO signals.
On J2SE, we use the NIO package to watch sockets using a mechanism similar to `select`, except that it is not possible to watch files, so on this backend IO signals for file ports are not supported.

7.4.3 The timeout signals

The timeout signals start the countdown to their emission when they are created. To that end we store the absolute time at which we started the countdown in each timeout signal. When the REL decides to query the readiness of IO signals (for backends which support it), it will call `select` or its Java NIO equivalent. Both of these functions take a timeout as parameter. The value of this timeout is the shortest interval before we have to emit a timeout signal (unless various conditions are met which we discuss in 7.4.5). This ensures

that if we start waiting for IO events to happen, we will stop waiting as soon as we have to emit timeout signals. If there are timeout signals which are already elapsed when we call `select`, we call it with the special timeout value telling it not to wait for IO signals and just report the current readiness values.

7.4.4 The idle signal

The idle signal is determined after the other REL signals, because it is only emitted when there are no other threads to schedule. This means that we have to look at the effect the other REL signals have on waiting threads: if the other REL signals did not wake any thread, and no other thread can be scheduled, we can emit the idle signal.
Note that it makes sense to consider (and emit) the idle signal only if it is being awaited, otherwise we may spend many instants emitting a signal which has no effect instead of passively waiting for agents or REL signals which would have an effect. If the idle signal is considered, it has the effect of not causing `select` to wait.

7.4.5 Integration with the EOI

The REL is considered at the beginning of instant, after threads have migrated and immigrated. At this time the scheduler already knows if there is any thread to schedule. When the REL is asked to determine the set of emitted REL signals, it will be instructed not to wait if there is any thread to schedule. To summarise, the REL will not wait if any of the following applies: there is any thread to schedule, there is an idle signal awaited, there is any elapsed timeout. If asked to wait, it will set a timeout determined by the timeout signals if there are any, or it will wait indefinitely for IO events.
Note that the addition of the REL cannot interfere with the various scheduling optimisations we described previously. This is because REL signals are considered at the beginning of instant, a phase which is before any thread has been scheduled yet, and the only thing the REL does during this phase is emit signals, which works exactly as if a thread was emitting them since we have already started the new instant.
It may happen that there are no REL signals, and no thread to schedule. In this case the VM has an option to terminate. But the default behaviour is to wait for incoming agents for further action. Because it is also meaningful to stop waiting for REL events in the case of incoming agents, the wait for incoming agents has to be integrated with the REL wait.
ULM VMs which await incoming agents asynchronously to the scheduler have to be able to stop any REL wait in progress to notify the scheduler of incoming agents. This is done in the case of `select` by writing on a pipe which is always watched by the REL, in order to notify `select`. For Java's NIO it is done by calling the `wakeup` method of the object on which `select` was called.
Although the J2ME VM does not have the equivalent of `select`, the REL is able to provide timeout and idle events using `sleep`, and incoming agents are awaited in asychronous threads which notify the REL's `sleep` by interrupting it with `interrupt`.

7.4.6 Future improvements

On the native backend, POSIX's `select` call is known not to scale well to large number of connections. On several systems a better, faster, more scalable approach exists: `epoll` on Linux, *kqueue* on FreeBSD or */dev/poll* on Solaris. We think that making use of these

mechanisms instead of `select` would be straightforward for ULM, and it would actually allow ULM to benefit from all their features. For example, it would be easy to register in the set of watched file descriptors when creating the IO REL signal instead of rebuilding the set for each `select` equivalent. Note that `epoll` is already used by J2SE internally as of version 6.
There are other external asynchronous events which may be interesting for ULM to integrate, such as POSIX signals. POSIX signals are usually integrated in an event loop by associating them an internal pipe to notify any pending `select` call like we do for incoming agents. This is one way ULM could implement POSIX signal integration. An simpler way seems to be in the works at least on Linux, in the form of `signalfd`: a new system call which maps a POSIX signal emission to an event on a file descriptor which can be used in `select`/`epoll`. This would be perfect for integration in the REL.
The work on integrating asynchronous events and event loops on Linux may also improve ULM's handling of timeout signals. A new system call `timerfd` maps a timer event to a file descriptor. The way the REL handles timeout signals is by calculating the closest timeout on each call to `select` and again calculating all those which have been triggered after the select call. Using `timerfd` and `epoll` it would be much easier and more efficient since all the timeouts would be handled by the kernel.

7.5 Integrating two event loops

Although the REL makes it possible to build programs based on event loops while enhancing them with the expressiveness of threads, we found we are still faced with the problem of having to interact with third-party event loops.
Short of writing our own GUI library from scratch and using the REL (or not) as the main loop of our library, we have to use third-party GUI libraries from ULM in order to display graphical interfaces for our programs. This is done by writing a wrapper layer to these libraries using ULM's native interface. But when doing that we also have to deal with their own event loop, and how it should integrate with the REL.
This is a common problem when having to integrate two event loops which both presuppose they are the one and only *main loop*. Should the REL be executed in the idle callback of the other event loop? Should the other event loop be executed in the idle instants, or at every instant from the REL? Should we try to integrate the two by merging one another's events in one of the loops? Or should we simply have both loops coexist in separate threads?
Bigloo's version of the FairThreads has a similar problem with its Biglook [GS02] GUI library, and they decided to invoke the FairThreads scheduler within the idle callback of their backend GUI libraries (GTK and Swing). Although it is not clear whether there is a better alternative, it is clear that this is far from optimal, since the idle callback is always executed, even for instants when there are no threads to schedule, thus wasting CPU cycles constantly.
In ULM, because we have a variety of backends for our VMs, we have had to interact with a number of different GUI libraries. Our Bigloo VM uses Biglook which in turn uses GTK for the native backend, or Swing for the Java backend. Our J2SE VM also uses Swing, while the J2ME VM uses the MIDP set of widgets. With this variety of GUI libraries, each using their own version of an event loop, we have tried every possible way to make their event loop and our REL interact nicely. We will present each GUI's specificity and the method used to integrate both loops for each case.

7.5.1 GTK

GTK is the GUI library used by Biglook for our Bigloo VM's native backend. It has at its core an event loop reified by an API allowing programmers to add new types of events (aside from IO, idle and timeout events). It is possible to create a new type of event representing the ULM scheduler: it would comprise the REL events as well as the incoming agents pipe which would allow us to wait passively for REL events or incoming agents when there are no threads to schedule. With such an *ULM scheduler event* registered in GTK's event loop, the loop would wait passively for things to do, and when it deems that there is work to do in the ULM scheduler, it would invoke our scheduler function in order to schedule one instant. Scheduling one instant at a time seems a good place for cooperation, because one event loop iteration would coincide with an instant.
Our VM and scheduler, however, have not been written with the intent to schedule only one instant then return. Once invoked, our VM expects the name of a main module, then enters various loops and functions in order to execute this main module's toplevel, main function and schedule every other thread until termination when there are no more threads to schedule and no REL event and incoming agent to wait for. It would require a lot of work to change that[2], so we decided to try another way.
Instead of calling the ULM scheduler from the GTK event loop, we decided to do it the other way around. We create a ULM thread in charge of executing one iteration of the GTK event loop at every instant if there is something to do in the GTK event loop. Determining when the GTK event loop has work to do is not obvious, because the API it provides does not allow introspection of registered events, which we could otherwise simply integrate as REL events. The main IO event registered in the GTK event loop is the file descriptor of the X server, which provides keyboard and mouse events to GTK. For the basic GUI programs we were writing, these are the only meaningful events we expect to be registered in the GTK event loop, so we decided to execute one iteration of the event loop every time we had incoming data on the X server's file descriptor (accessible from the standard GTK API) using a REL IO signal.
By using this system we are sure that even callbacks registered for example for a button widget, would be called from within the ULM thread executing the GTK event loop. Of course with this system we will not call the GTK event loop for events other than the mouse and keyboard events, but should we need to, we believe it would be easy to add the introspection facilities to retrieve the list of events and integrate them in the REL.

7.5.2 Swing

Swing is the GUI toolkit used by J2SE, therefore it is used by Biglook's Java backend for our Bigloo VM, and by our J2SE VM. Getting the Swing event loop to be executed within a ULM thread is not possible because its API does not allow this. Based on the available API we found a way to both detect incoming Swing events and block the execution of its event loop after every iteration.
Because it is possible to register a callback to be executed on the event loop, we created a callback which blocks the event loop until it detects a new event to process. When it does detect a new event, it notifies the REL using an IO signal. A ULM thread waits for this signal to be emitted and orders our previous callback to unblock the event loop by returning to it. We then add the callback again at the end of things to execute on the event

[2]Well, actually changing it is never the problem, getting it to work again once changed is the challenge.

loop in order to block it again, and awake the ULM thread which was waiting for the event loop to be flushed.

Since the event loop keeps every event (and callbacks) in a FIFO we are guaranteed that by returning from the blocking callback and registering it again, every event and callback registered between the two will be executed by the event loop.

While this does not execute the Swing event loop from within a ULM thread, they are synchronous to each other, which means we are able to treat any callback executed by the event loop flushing (like button pressed actions) like a normal reentry from the native stack as described in the previous chapter.

7.5.3 J2ME

The graphical toolkit in J2ME is very different from Swing, but the only difference for us is that the event loop is not accessible at all. We know it is there, but there is no way for us to access it, so we found no way to block it. On this platform we decided to let the ULM VM and the GUI run in separate asynchronous threads and try something new. From a ULM thread we can create and manage the GUI, provided it is only done from within ULM threads, it is like sending a synchronous flow of modifications to an asynchronous thread (the GUI's event loop) which supports asynchronous modifications.

The execution of callbacks from the GUI on the other hand, cannot result in the execution of ULM code from within the GUI's event loop, otherwise we would have two asynchronous threads executing the ULM VM. So we decided to introduce an asynchronous delivery of ULM signals. If the GUI wishes to notify the ULM threads of an action (a button press), it instructs the ULM scheduler to emit a signal, like every other external source of events, at the beginning of the next instant.

For example, when a ULM thread creates a button, it creates a signal, which it associates to the button's press action. The thread can then wait for the signal, which will be emitted by the GUI which will emit it asynchronously via the ULM scheduler.

This method guarantees that modifications to the GUI will be done synchronously, since they are done in ULM, and that callback functions will be invoked in ULM threads correctly.

The only problem we found with this solution is when the user presses two buttons shortly after one another, thus queueing two asynchronous signals which will may be emitted at the same ULM instant, thus causing the execution two callbacks which may not be the expected behaviour (for example in the case of a "OK" or "Cancel" dialog, we expect only one choice to be made before the dialog disappears). In order to avoid that we can limit the emission of asynchronous signals to one per ULM instant, but this only guarantees that each asynchronous signals will be sent in sequence in different instants. This does not guarantee that the first event's callback will remove the dialog window in order to prevent the user from sending the second event.

In an (single) event loop this is not possible: the first button press executes the callback instantly, so if the dialog is then removed, the user cannot get enough time to press the second button. This is because callbacks executed from the event loop effectively block the handling of mouse and keyboard events, thus delaying the mouse click which would cause the second button to be pressed until the button is gone.

7.6 Conclusion

With the Reactive Event Loop, we think it is easier to build more powerful event-loop-based programs than with traditional event loops, thanks to the addition of cooperative threads. The implementation of REL signals is already efficient but there is still room for improvement. Unfortunately, short of building a GUI entirely in ULM, it is still relatively hard to integrate efficiently the ULM scheduler with a third-party event loop such as comes with GUI libraries.

Chapter 8

Examples/Applications

However beautiful the strategy, you should occasionally look at the results.

– Winston Churchill

Over the course of developing ULM, we have used ULM in several applications, ranging from the ludicrous chase of fox and rabbit agents [Epa04] to more down-to-earth applications which could be used for automatic reconfigurations of internet boxes. In this chapter we will describe two ULM applications: the first one is academic and serves both as an example and reminder of every ULM feature, while the second is a more industrial example of the usefulness of ULM.

8.1 Load balancing

Load balancing, the act of distributing computing costs across a network of computers, is a popular use for agent mobility [MMM02] [SSCJ98] [CSWD03]. In this section we will present the full implementation of a load balancer without focusing on the mathematics of balancing. We want to create a load balancer that will take a thunk and executes it by sharing the computation cost across the network. Because this is a simplified example, we will say that when there are too many such functions executed at once by our load balancer, some of the computations are going to be sent to another site for completion.
The basic idea, of course, is to use agents to execute these computations, and create a custom scheduler which will either allow these agents to spend an instant computing (using suspension), or send them to another site (using migration). In order to start the balanced computation of a thunk, we will have to create an agent which will execute this thunk in a suspension context. This agent will then be registered within our load balancer scheduler (a simple thread) which will decide which agents to unblock for one instant (by emitting their suspension signal) and which agents to send away.
Because we want to be a little bit fair in how the agents are balanced, we will attribute a counter to each agent, incremented when they are sent to another site, so that the agents that have been migrated the most often will get a better chance of being executed.
Since we need to associate several properties with our agents, it makes sense to create a mixin type for our agents, where we can store properties such as the agent's name (used for objective migration), migration counter and suspension signal.
Let us start by looking at the scheduler. We define a variable holding the maximum number of computations running during an instant:

```
; the maximum number of agents running at once
(define max-agents 5)
```

Then we define a variable to hold the list of balanced agents, and two functions to register and unregister an agent to our load balancer:

```
; the list of regulated agents
(define agents '())

; register a regulated agent
(define (lb:register a)
 (set! agents (cons a agents)))

; remove a regulated agent
(define (lb:unregister a)
 (set! agents (remove a agents)))
```

Our custom scheduling function will be invoked by our load balancer at each instant. Its task is to select the first `max-agents` agents that have been delayed the most in order to emit their suspension signal so they can be executed during this instant. The remaining agents will have their `delay` field increased, and will then be sent to a random site for balancing. You may notice that we also emit an additional signal for those migrated agents: this is used to awaken an auxiliary thread associated with each agent. This thread will wake up in order to unregister the agent from this load balancer, and register it in the destination site's load balancer. This is required because the agent is suspended in the middle of a computation and cannot accomplish this task.

```
; the custom scheduling function
(define (lb:schedule)
 ; sort them with the most delayed first
 (let* ((agents (sort (lambda (a b) (> a.delay b.delay)) agents))
        ; take a sublist of max-agents length
        (to-run (slice agents 0 max-agents))
        ; and those we delay
        (to-delay (slice agents max-agents (length agents))))
  ; emit the suspension signal for each scheduled agent
  (for-each (lambda (t) (ulm:emit t.susp)) to-run)
  ; increase the delay counter for those we delay, and migrate them
  (for-each (lambda (t)
              (set! t.delay (+ t.delay 1))
              ; send it to a random site
              (ulm:migrate-to (random-site) t.name)
              ; wake its auxiliary thread up
              (ulm:emit t.migrate))
   to-delay)
))
```

The only thing left to do for our custom load balancing scheduler is to start a thread which will balance the load at each instant:

```
; create the scheduler thread
(ulm:thread
 (let loop ()
  ; schedule them at each instant
  (lb:schedule)
  (ulm:pause)
  (loop)))
```

Our agent mixin will contain several field we have already seen, such as the suspension signal, the delay counter, the agent name, and the auxiliary thread's signal. The mixin constructor will take only one argument: the thunk to balance.

```
; the balanced function mixin
(define-mixin (lb:balanced-function thunk)
 ; this is the suspension signal
 (var susp (ulm:signal))
 ; this is the delay counter
 (var delay 0)
 ; add a signal for the auxiliary thread
 (var migrate (ulm:signal))
 ; this will hold the name of the agent
 (var name #f)
```

We also need to store several other fields in our mixin: the origin site of the agent, so that once the balanced computation is over, the agent can return back to its origin site to communicate the return value of the thunk. That return value will also have to be stored so it can be returned: we will store it in a reference because it has to be affected by the agent, and read by the thread which ordered this balanced computation. Because the agent can migrate, the only way to share a variable between agents on different sites is to use a reference. In order for the thread which ordered the balanced computation to wait for the result, we use another signal `done` which will be awaited by the thread, and emitted by the agent once it is done and back:

```
 ; store our origin site
 (var home (ulm:current-site))
 ; store the computation result in a reference
 (var res (ulm:ref #f))
 ; this is emitted when the computation is done
 (var done (ulm:signal))
```

We can now define the `start` method of our agent, which will start an agent, start its auxiliary thread, register in the load balancer, then execute the thunk in a suspension context. When the computation is done, the agent migrates back to its origin site, stores the result in the `res` reference and notifies its completion by emitting the `done` signal:

```
; override the method which starts the computation
(meth (start)
 ; start in an agent
 (ulm:agent (name)
  ; start the auxiliary thread
  (this.start-auxiliary-thread)
  ; store the agent name
  (set! this.name name)
  ; register it in our custom scheduler
  (lb:register this)
  ; execute the thunk in the suspension context
  (let ((res (ulm:when this.susp (thunk))))
   ; unregister the regulated function
   (lb:unregister this)
   ; go back home
   (unless (equal? (ulm:current-site) this.home)
    (ulm:migrate-to this.home)
    (ulm:pause))
   ; we can now store the result
   (ulm:ref-set! this.res res)
   ; notify completion
   (ulm:emit this.done))))
```

The auxiliary thread consists in a thread created by the agent, and therefore linked to it: this thread will be migrated everywhere with the agent. This thread will loop infinitely waiting for the scheduler to emit the `migrate` signal, so that the thread will unregister the agent from the local load balancer, pause to get to the destination site, and register the agent in the new load balancer. Because we want this thread to stop existing when the agent is done, we execute this loop in a preemption context on the `done` signal, which is emitted by the agent upon completion:

```
(meth (start-auxiliary-thread)
 (ulm:thread
  ; stop doing all that when the agent is done
  (ulm:watch (ulm:unref this.done)
   (let loop ()
    ; await a migration signal
    (ulm:await this.migrate)
    ; unregister locally
    (lb:unregister this)
    ; wait for the objective migration to occur
    (ulm:pause)
    ; register on the new site
    (lb:register this)
    (loop)))))
)
```

Our final function is the function which starts a balanced computation. Calling this function with a thunk would cause this thunk to be balanced until it has emitted the `done` signal

and set its result value `res`. This function creates a balanced function mixin with the given thunk, calls its `start` method and waits for its result, which it then returns:

```
(define (lb:run thunk)
 ; create a regulated function
 (let ((rf (new lb:regulated-function thunk)))
  ; start it
  (rf.start)
  ; wait for it
  (ulm:await rf.done)
  ; return its result
  (ulm:unref rf.res)))
```

We have seen the full code of the load balancer. The beauty of the above example is that we have seen every single feature of ULM: suspension, preemption, agents, thread groups, signals, references and mixins. What makes this approach original is how easy it is to create custom schedulers with suspension: the thunk we are balancing does not have to be written specially for load-balancing, it only has to be written so as to cooperate. The point is that any function written for local threads would be able to be balanced using this system.

8.2 Agents for reconfiguration

During the course of this thesis, we developed a proof of concept for France Telecom. The idea was to use ULM agents for automatic reconfiguration and delivery of software components. We will present the motivations which led to the use of ULM agents, and describe the two proof of concept applications which emerged as a result.

8.2.1 The motivations

The team at France Telecom we were in contact with worked on software components in the Fractal [BCS02] model. More specifically they worked on the dynamic reconfigurations of such components in several contexts. For example, France Telecom's internet subsidiary Orange rents an internet terminal called "LiveBox" to its subscribers. It would be interesting if the firmware of this LiveBox could be remotely upgradable through software component reconfiguration instead of by replacing the entire firmware at each upgrade.
There are several other contexts of use which they described through scenarios such as the "traveller scenario". It describes the journey of a traveler from his car while going to the airport, until his arrival in a new country, as it relates for his *mobile terminal*. This term refers to any mobile device used by the traveler, such as a mobile phone, smart phone, PDA or laptop. In this scenario, upon arrival at the airport, the mobile terminal would indicate parking availability and fares to the user. Within the airport terminal the device would offer to do the flight check-in directly on the mobile terminal by choosing the flight number and seats. Finally, upon arrival on the destination country, the mobile terminal would check for any software components available which might be needed for this country, such as a local communication protocol.
The "traveller scenario" may seem simple when described in such broad terms, but it actually involves many complex steps and techniques. In our example, the mobile terminal starts by receiving a simple message from the airport's parking. The airport terminal then

asks the user a question: does he want to check-in via his terminal ? This involves more than simple interaction between the airport and the user's terminal: if the user chooses to check-in, the check-in program will have to be downloaded and installed. Presumably there could already be an older version of that program installed, which might have to be upgraded. This program will then have to communicate with the airport to fulfill the check-in process. Upon arrival, the mobile terminal will ask the airport for a list of required components, download, install and configure them.
There are many ways to implement solutions for these requirements, but we have chosen to concentrate on two phases of this scenario with different uses for ULM agents in each phase. The first phase is the interaction with the departure airport. We have chosen to show how easy it is for the airport to interact with the user using agents. The second phase is the arrival, or more specifically the task of finding, transporting and installing new components. In fact, in order to make the second phase more appealing in the absence of GSM software components to transport on a device, we have chosen a different scenario with similar requirements: an IRC (Intenet Relay Chat) client on the mobile terminal which reconfigures its encryption components on each site it visits.

8.2.2 The departure airport

For the first phase of this application, we have chosen to demonstrate the usefulness of ULM agents in the settings of a server (the airport) which wants to interact with a client (the mobile terminal) which does not know how to interact with the server prior to their encounter.

Choosing the mobile device

Ideally, we would have chosen a full-fledged remote terminal for this application, such as the Nokia 770 used by France Telecom, but several limitations forbade us from doing so. Indeed we have to be able to run an ULM VM on the device, and we were unable to do so. We support J2ME, J2SE, C and .NET systems, but the Nokia 770 in question has very limited beta support for both J2ME, J2SE and .NET virtual machines. As for C, building a version of our ULM VM for this system requires the use of the QEmu emulator for ARM devices in order to cross-compile our VM. But our VM uses asynchronous threads internally, and QEmu does not support threads when emulating a process on an ARM processor on our development machine. This, the very limited support and the fact that many development solutions for this device are in their early stages meant that we would spend more time getting ready for the device than solving problems with ULM.
We have therefore decided to use the Nokia phone we used to develop the J2ME ULM VM for this application. This meant that the ULM VM would be fully functional, although our support for Fractal components would be limited. Indeed, the implementations of Fractal in Java (Julia [BCL+04] and AOKell [SPDC06] for example) require more features than available in J2ME to dynamically load components. Indeed, J2ME platforms support either of two sets of APIs: the CLDC (Connected Limited Device Configuration) and CDC (Connected Device Configuration) profiles. The CLDC profile is available on virtually every mobile phone and many mobile devices, but does not include the Java `ClassLoader` class, which is required to load class dynamically in Java. Without this class it is impossible to load a class from a Java VM if that class was not available from the start of the VM, which means we cannot install classes after the VM's start. The CDC profile on the other

hand, features this class, but this profile seems to have been forgotten over the years and we could only find one single phone supporting it, which dates from several years ago and is not available anymore.

In the absence of dynamic class loading, it is impossible for Java implementations of Fractal to load components dynamically. We do not have this problem in ULM because we load agents without using dynamic Java classes, but we do not have any Fractal implementation in ULM, and while it may be possible to write one, it was not the purpose of this work.

On the other hand, the scenario we were trying to solve presented an interaction between the server (airport) and client (mobile terminal) which we found we could solve easily in ULM without the need for any Fractal components.

The interaction program

In this application, we phone client does not have any prerequisite program or library on it aside from the necessary GUI library. All the logic is initiated and served by the server airport. This server runs on a PC in the Bigloo ULM VM. The first step the server has to accomplish is to detect any new client. This can be done simply because we communicate with the client using Bluetooth, which supports scanning the nearby devices. Using the REL and a native call which lists the nearby devices, we are able to write a crude detection mechanism:

```
(module airport
 (import srfi-1 gui reserv)
 (native (bluetooth ((bigloo . bluetooth))
          (list-devices))))

; check every second
(let ((sig (ulm:timeout-signal 1 0)))
 (ulm:when sig
  (let loop ((old-devices '())
             (new-devices (list-devices)))
   ; serve the newly detected devices
   (for-each serve (lset-intersection equal? new-devices old-devices))
   (ulm:pause)
   (loop (lset-union equal? old-devices new-devices)
         (list-devices)))))

; continued...
```

Of course a real system would be more complex and would only detect devices coming by car (not by plane), and would clean the list of detected devices when the user leaves the site for example. But this will be enough for our example. The `serve` function consists in sending an agent to the device to notify the availability of parking space, then ask if the user wants to check-in, and if yes, start the check-in procedure. Here it is:

```
; ...continued

; shortcut function for migration
(define (move site)
 (ulm:migrate-to site)
 (ulm:pause))

(define (serve device)
 ; send an agent there
 (ulm:agent (name)
  ; go there
  (move device)
  ; from then on, we are on the phone
  (gui:message "Available parking on the 4th floor")
  ; offer the check-in
  (if (equal? "Yes"
              (gui:question "Do you want to check-in?"
                            '("Yes" "No")))
   (do-checkin))))

; continued...
```

Naturally, a real system would actually have external input as to where there is available parking. The `gui` module consists of several GUI functions available in a ubiquitous module which is actually implemented on the phone itself. Its interface (on the server) is as follows:

```
(ubiq-module gui
 (export (gui:message text)
         (gui:question text choices)
         (gui:input text)))

; Shows a message and returns when the user
; has clicked 'OK'
(define (gui:message text) #t)

; Asks a question, and displays a list of choices.
; The selected choice is returned
(define (gui:question text choices) #t)

; Asks the user to enter some text, returns it
(define (gui:input text) #t)
```

The `do-checkin` procedure consists in asking the reservation number, then going to the reservation server (which may be different than the site which sends these agents) to invoke the `reserv:checkin` function. This function takes at least one argument: the reservation number. If the reservation is invalid or unknown, it returns a pair whose `car` is `err` and whose `cdr` is the error description. If the reservation is valid and there is only one seat left it returns a list whose `car` is `seat` and whose remaining elements describe the seat and flight. If the reservation is valid but there are several seats to choose from it returns a

list whose `car` is `seats` and whose rest contains the list of available seats. Once the user has chosen a seat, the agent has to re-invoke the `reserv:checkin` method with the chosen seat as additional argument. The `reserv` module is ubiquitous and its implementation is located on the reservation site whose address is in the `reserv:site` variable. Here is the `do-checkin` procedure:

```
; ...continued

(define (do-checkin)
 ; save the device's address
 (define device (ulm:site-address))
 ; ask the reservation number
 (let ((number (gui:input "Reservation number:")))
  ; go to the reservation site
  (move reserv:site)
  ; try to check-in
  (let loop ((res (reserv:checkin number)))
   ; move back to the device with the answer
   (move dev)
   (case (car res)
    ; display the error message and start over
    ((err)
     (gui:message(cadr res))
     (do-checkin))
    ; success: display information
    ((seat)
     (gui:message (string-append
                   "Name: "
                   (list-ref res 1)
                   "\nFlight: "
                   (list-ref res 2)
                   "\nDescr: "
                   (list-ref res 3)
                   "\nSeat: "
                   (list-ref res 4)
                   )))
    ; we have to chose a seat
    ((seats)
     (let ((seat (gui:question "Choose a seat" (cdr res))))
      ; check-in again with the chosen seat
      (move reserv:site)
      (loop (reserv:checkin number seat)))))))))
```

We will not show here the contents of the `reserv` module since its content is neither good enough as a reservation system, nor relevant to the use of agents for this situation. Its interface is as follows:

```
(ubiq-module reserv
 (export reserv:site
         (reserv:checkin num . seat)))

(define reserv:site "...")

(define (reserv:checkin num . seat) #f)
```

Conclusion

While this program is only a proof of concept, it shows that it is relatively easy and unexpensive in terms of lines of code to write applications in ULM which can interact with devices without prior support from the device. Using traditional client/server architectures on mobile phones, the user would have had to browse the airport's website to find the check-in program, install it, then run it to get the same result. On the other side, the would have had to write two programs: one which runs on the client, the other which answers the client's requests.
Even if the airport decided to use a check-in program on its website (in the so-called Web 2.0 for example), the user would have had to be informed of the availability and location of this website. Of course, accepting any agent on our mobile phone is not a perfect solution either, and there should be a certain level of trust. Currently we use a system of trusted signing certificates to sign outgoing agents and accept incoming ones. On top of that, a simple question from the ULM VM to the user would suffice to know if the user is interested in a trusted agent.

8.2.3 The Fractal component transporter

In order to demonstrate the possibilities of ULM to manage the dynamic reconfiguration of components, we have implemented an IRC (Internet Relay Chat) application using Fractal components.

The design

The problem we want to solve with ULM is the complex management of reconfiguration and component selection. To that end, we designed an IRC clone which uses Fractal components to encrypt the communications between the client and the server. The IRC client is supposed to be on a mobile terminal which can move between several sites (for example: home, office, area 51...), each with different requirements for security and encryption. This IRC client, upon arrival on a new site, would send a ULM agent on the site's server to ask for the address of the local IRC server and a list of available encryption components.
Any communication with the local IRC server would have to be done using one of the components listed locally. If the component is already installed on the IRC client, it is then used for encryption when connecting to the IRC server. In the case that the client does not have the component, the agent would use information gathered from the IRC client to find a component compatible with the mobile device. If such a component is found, the agent would transport its package back to the mobile device to install it so that the IRC would be able to use it from then on.

We used the Julia implementation of Fractal components and the J2SE ULM VM. Our initial intention was to use a mobile device such as a PDA or a smartphone, but as described previously our attempts at running a JVM capable of loading classes dynamically on these devices have been unsuccessful. In any case this does not deprecate our example as we can use the IRC client on a laptop to transport it between sites.
The type of encryption component we used is called `SecurityComponent` and consists of two methods for encryption and decryption (`encrypt` and `decrypt`) and a method used to determine the component's compatibility with a given Java implementation (J2ME, J2SE...) and version. We simplify the compatibility issue by basing it on the type and version of Java on the client, but many other factors could come in play such as the client's FLOPS (Floating Point Operation Per Second) or amount of RAM. The encryption is also overly simplified and does not talk about algorithms, keys and other necessary features since it was not the point of this example to implement full-blown Fractal encryption components. Any implementation of such a component is then packaged into a JAR with a meta-information file which lists the available component interfaces and their implementing classes available in the JAR. On each site there is a `ComponentManager` object which can load such JARs dynamically and register any interface and their implementations for later instantiation and use. This manager also keeps track of the JAR size, which is used by the agent to determine whether there is enough room on the client to install the component[1].
We have implemented and packaged several encryption components with various requirements for Java support and varying JAR sizes, and have distributed them across the sites so that the each site offered several encryption components to choose from, while each site had a different set of unique components. Indeed it is unlikely that the user would need a military-grade encryption component at home for social chatting.

The program

Aside from the fact that the client uses a Fractal component of the `SecurityComponent` interface, there is little interest in presenting the many lines of code which comprises the building and handling of the GUI. This GUI presents a site selector, which will initiate the connection and configuration on the specified site. It also contains a list of values to choose from in order to simulate client constraints such as Java type and versions, available disk size, etc... There are an entry for the user to type text, one where he sees the text sent from the server (from the other users), as well as a description of the security components currently installed and used.
Whenever the user selects a new site, an agent is launched to that site. Because all the GUI is done in Java we have decided to leave the Java event loop and the ULM scheduler in separate asynchronous threads. In order to invoke ULM code from the native (Java) layer, we use a new technique different from those presented in the REL chapter. The idea is that we create a native module called `Util` which has a Java method called `invokeULMFunction` which takes as arguments a ULM closure and its list of arguments. This function will spool those invocation requests and emit an asynchronous signal on the ULM VM. This signal is awaited by a normal ULM thread, which will then collect the spooled invocation requests, and treat each one in a new ULM thread. When the function returns, the ULM thread will notify the result to the caller of `invokeULMFunction` which will then return the collected

[1]This attribute was preferred over FLOPS or RAM numbers because on our J2ME phone we know the disk space but ignore any FLOPS or RAM capabilities.

```
// called when selecting a new site
public void register(String site, VirtualMachine vm){
  // wait for the asynchronous VM to load the native Util module
  Util vmMod = (Util)vm.awaitModule(Symbol.getSymbol("util"));
  // wait for the asynchronous VM to load the component-agent ULM module
  Module agentMod = vm.awaitModule(Symbol.getSymbol("component-agent"));
  // obtain the register closure
  Closure f = (Closure)agentMod.getGlobal(Symbol.getSymbol("register"));
  // build the list of arguments
  Cons args = Cons.listify(new Object[]{site, "SecurityComponent",
                                        this.availableSpace,
                                        this.javaType,
                                        this.javaVersion});
  // invoke the function in a ULM thread
  Object ret = vmMod.InvokeULMFunction(vm, f, args);
  // check for failure
  if(ret == R5RS.CONSTANT_FALSE){
    // treat the failure
    notifyFailure();
  }else{
    // connect to the IRC server using the specified component
    String server = ULM.ULM_TO_STRING(((Cons)ret).car);
    String implementation = ULM.ULM_TO_STRING(((Cons)ret).cdr);
    useComponent(implementation);
    connectTo(server);
  }
}
```

Figure 8.1: The invocation code

result.
When selecting a new site, we use the ULM VM API to load the `Util` module and the `component-agent` module which contains the agent code. This agent is sent by the `register` ULM function which expects as arguments the address of the site, the name of the component interface to fetch and the available size and versions of the local Java. In case of success, it returns a pair whose `car` is the address of the local IRC server and whose `cdr` is the name of the security component implementation to use. In case of failure it returns `#f`. The code of the invocation is shown in Figure 8.1.
The ULM module which interfaces with the local `ComponentManager` has the interface shown in Figure 8.2.
This module is then used by our agent's module, defined in Figures 8.3 and 8.4.

Conclusion

While this application is a simple proof of concept, it shows that ULM agents make it possible to devise complex component-selection mechanisms. We can imagine that the agent could check for suitable components on several component servers, should the first ones fail

```
(ubiq-module component-manager
 (export (cm:list-component-impls interface)
         (cm:get-jar interface impl)
         (cm:get-jar-size interface impl)
         (cm:instantiate-component interface impl)
         (cm:isJavaSupported c javaType javaVersion)
         (cm:install-jar data)
         (cm:get-irc-server)
  ))

; lists the names of every registered component class
; which implements the given interface
(define (cm:list-component-impls interface) ...)

; gets a component packaged as a JAR in a vector
(define (cm:get-jar interface impl) ...)

; gets the component's package's size
(define (cm:get-jar-size interface impl) ...)

; instantiates a component
(define (cm:instantiace-component interface impl) ...)

; invokes the isJavaSupported method on the given
; SecurityComponent instance
(define (cm:isJavaSupported c javaType javaVersion) ...)

; installs the given jar in a vector and registers every
; component interface and implementation inside
(define (cm:install-jar data) ...)

; gets the address of the local IRC server
(define (cm:get-irc-server) ...)
```

Figure 8.2: The component manager interface

```
(module comp-agent
 (import component-manager srfi-34)
 (export (register server interface size jtype jversion)))

; returns #t if the given component implementation supports
; the given Java and fits in the given size
(define (fit-finder interface impl size jtype jversion)
 (let* ((c (cm:instantiate-component interface impl))
        (fit (cm:isJavaSupported c jtype jversion))
        (csize (cm:get-jar-size interface impl)))
  (and (<= csize size) fit)))

; finds the component with the given interface on the server:
; if the server lists an available component which we already have in
; here-components (the list of components available on the client), it is used.
; If not we try to find an acceptable component using fit-finder.
(define (get-component interface here-components size jtype jversion)
 ; get the list of components available on the server
 (let* ((there-components (cm:list-component-impls interface))
        ; find any component already available on the client
        (old (find (lambda (impl)
                     (member impl here-components))
                   there-components)))
  ; if we found one, return its name and the IRC server
  (if old
   (list 'old (cm:get-irc-server) old)
   ; if not, try to find a fit component
   (let ((new (find (lambda (impl)
                      (fit-finder interface impl size
                                  jtype jversion))
                    there-components)))
    ; if we find it, return its name, its JAR and the IRC server
    (if new
     (list 'new (cm:get-irc-server)
           new (cm:get-jar interface new))
     #f)))))

; continued...
```

Figure 8.3: The reconfiguration agent (1)

```
; ...continued

; registers on the server to find a suitable component implementation
(define (register server interface size jtype jversion)
 ; use an exception handler if the migration fails
 (with-exception-catcher
  (lambda (exc)
   (print "got exception: " exc)
   #f)
  (lambda ()
   ; list the client-side components
   (let* ((here-components (cm:list-component-impls interface))
          ; this is the safe-migration RPC which calls the given function
          ; on the server site
          (ret (ulm:rpc
                server
                (lambda ()
                 (get-component interface here-components
                                size jtype jversion)))))
    (case (car ret)
     ; we found a previously installed component
     ((old)
      ; ret := (old cm-server impl)
      (cons (cadr ret) (caddr ret)))
     ; we found and brought a new component
     ((new)
      ; ret := (new cm-server impl data)
      ; we need to install it
      (let ((impl (caddr ret))
            (data (cddddr ret)))
       (cm:install-jar data)
       ; return
       (cons (cadr ret) impl)))
     ; we did not find anything
     (else
      #f))))))
```

Figure 8.4: The reconfiguration agent (2)

to produce suitable components. We can also imagine that in the lack of appropriate component, the agent could come back to the client with a server-provided program (or agent) which would handle the problem on the client side, via interaction with the user. This way every site could have its own policy and methods of dealing with incompatible clients.

Chapter 9

Directions

> *Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.*
>
> – Winston Churchill

Throughout this dissertation, we have described the state of the ULM language, compiler and virtual machines. In the process we have sometimes talked about various future modifications or improvements we would like to see, such as ubiquitous values or custom serialisers. There are other improvements we think would be valuable, although we have not discussed them yet because they were not directly relevant to the previous chapters. In this chapter we will discuss these various missing features we have identified and start the discussion on what might be missing.

9.1 Debug

When developing a new compiler and virtual machine, the very first thing one notices when trying the first test programs is that both the compiler and the virtual machine are full of bugs. Debugging a compiler and a VM is a hard task which is very different from debugging other types of programs.
For example, the crash of the VM may indicate a bug in the VM itself, or in the compiler which produced faulty bytecode. In order to ensure that a given ULM source file has been compiled to a correct bytecode, one has to have the (rare) ability to read the bytecode, make sense of it and come to the conclusion that it is a correct compilation. Even with the use of automated test systems which compare source files and their resulting bytecode, one has to certify the first correct bytecode which will serve for comparison.
Once we have ascertained that the compiler is not to blame, debugging the virtual machine is no simple task as we have already explained it consists at its core of a loop which never recurses and holds no information in its stack[1]. Traditional debuggers help programmers by showing the location of a program crash by inspecting its stack to show the succession of function calls which (possibly) led to the crash. In our case, any posthumous inspection of the VM stack will yield absolutely no information on the events which led to the crash. This information will have to be found elsewhere.

[1] And indeed our VMs are compiled into programs which use a stack for continuations.

After the tremendous task of certifying that the bytecode is correct and that the VM was correct in its interpretation, the only suspect to the cause of the crash (or misbehaviour) is the ULM program itself. Debugging the VM serves no purpose in this case, but the VM has to allow a debugger to inspect the running program's state so that the programmer can debug his ULM program. Note that this is different from debugging the compiler and the VM where it should be up to the VM and compiler programmer, not the casual ULM programmer, to debug these programs. If there are many people working on the compiler and VM, it is as important to have the proper tools to debug them: as important as debug tools for the language programmer, since a correct compiler and VM are crucial.

9.1.1 Debugging the compiler

Although our compiler is very simplistic in the sense that it does not perform any code optimisation, it is still complex enough to contain several bugs. These can easily be found in the primitive compilations during development or introduction of new features, the many macro expanders, or the various evolutions to the mixin system. The fact that the bytecode format also changes as we find unforeseen limitations and various enhancements led to the development of two separate tools to analyse an OULM file: the `ulmp` program which analyses an OULM file formatting, and a disassembler named `ulmd`.

The OULM analyser

There are many reasons why we would want to look at an OULM file to see what it contains. The main reason for us is to analyse it to make sure the OULM format is correct. Indeed there are many places in the code where such a file format is created: the compiler, but also the Bigloo and Java ULM VMs which produce such files when sending agents. Although the Bigloo VM and the compiler share the OULM-producing code, it is not uncommon when introducing a new version of the OULM file format to have discrepancies between the two OULM writers and their corresponding reader code.
The second reason why we would want to examine OULM files is when sending agents. Often it is useful to instruct the VM to dump the agent it sends to files prior to sending so that they can be examined posthumously in the case of failure after migration. This is often the only way to make sure that the serialisation process worked as expected and that every data which was supposed to be taken along with the agent was indeed correctly identified and serialised. For the casual ULM programmer it is also useful to identify problems such as the size of data taken by an agent, in order to add or remove ubiquitous references if the agent left with not enough or too much data.
In order to present the user with a useful view of OULM files, we have developed the `ulmp` program, which takes a OULM file as parameter and dumps the contents of this file. In fact the Example part of the description of the OULM file format in the Implementation chapter is directly taken from the output of the `ulmp` program.

The disassembler

The `ulmp` program displays the various fields of the OULM file format, including the bytecode displayed in a human-readable form by showing the bytecode names and pointing to their arguments rather than displaying the bytes themselves. This is very efficient to debug any problems in the file format or the encapsulation of agents and their data.

There are, however, reasons why we would prefer to inspect the bytecode content in a higher-level form such as ULM itself. The process of producing higher-level code from its lower-level compilation is called *disassembly*. In an agent system where agents can collect closures (and their code) and global values from any site, it is key to any *source-level debugging* (the process of debugging a program by looking at and reasoning from its source code).
Indeed, it is possible (and very likely) that the source code from which originated an agent on its departure site will be completely different from the agent code after a few migrations. The process we have chosen for migration makes a new module from each migrating agent. This involves the capture of any required functions (and their code) as well as global variables from any number of modules external to the agent itself. This new module itself may already have little in common with the code of the module which spawned the agent. After a few migrations where the agent may have captured new code and data (which may have been completely absent from the origin site) and possibly dropped unused code and variables, the resulting agent module may be completely unrecognisable.
Should this new code crash after a few migration, the debugger would be incapable of displaying its source code to the user. Short of carrying the source code of every procedure captured by the agent during migration, disassembly is the only solution to present the agent's source code during debugging. To that end we have written the `ulmd` program which takes an OULM file (such as produced during migration for debugging) and writes its corresponding ULM code.
The process of writing a disassembler is very similar to that of the VM itself: we have to loop over the bytecode and construct source code as we go instead of evaluating its result. Although the disassembler is a mere thousand lines of code (it does share some code with the Bigloo VM and compiler), it is a very interesting piece of machinery which mingles the VM iteration and the compiler structure to produce source code from a bytecode which was never intended to be reversed to its origin form. Unfortunately it is beyond the scope of this thesis to describe it in detail here[2].
We have successfully mapped every bytecode and every primitive compilation backwards from bytecode to source code. After the strict *uncompilation* phase we apply a phase of *unexpanding* to undo the macro expansions introduced during compilation. This macro unexpansion is done by a process described at best as *shaky* but largely functional. Indeed we are not able to unexpand user macros, but every macro expansion done by the compiler (including those which result from several previous expansions) is able to be reversed by our disassembler. With the compiler set to include the names of variables in the debugging information, the whole ULM runtime (about 2300 lines of code) is successfully disassembled to the exact source.

9.1.2 Debugging the VM

Due to the nature of the VM itself, the fact that it does not store state in its stack, it is largely debugger-proof. Indeed, using a source-level debugger is in most cases useless since a VM crash is closely tied to the bytecode it is executing, and in most cases debugging the VM requires debugging the ULM source at the same time.
We have found no particular technique to help us find bugs in the VM, particularly in the bytecode-execution loop itself. In fact once we have made sure the bytecode is correct using

[2]This is not unreasonable bearing in mind this dissertation's length.

the bytecode dumper and disassembler, and we have studied the ULM program enough to be confident that the VM is at fault, we turn on the debugging output on the VM and spend hours reading the output. Follows an intricate analysis of the step-by-step debugging output, along with a dump of the state of every VM structure (including ULM threads' stacks) and register, and careful analysis of the ULM program and its bytecode.

There are several classes of programs which require specific debuggers, and we believe this is one of them. We think the development of a debugger which could debug the VM and the ULM program it is executing at the same time would help in this regard. On the other hand, since the number and size of programs written in ULM is expected to be higher than the code required by the VM to run these programs, one can expect that it would be more useful to write a debugger for ULM programs and just make a functional VM using ad-hoc means.

9.1.3 Debugging ULM programs

Once we are confident that the compiler and VM produce correct OULM files, with correct bytecode and serialisation, and that the VM interprets them correctly, we are left with ULM programs to blame for any remaining bugs. We are confident that the disassembler would be key for any source-level debugger for ULM, but it is not sufficient. There are several aspects of ULM which would need unique features from its debugger.

The fact that ULM code can crash on a remote site after an unknown time and number of migrations causes problems is one specific aspect: a user wishing to debug an agent would need support from the debugger to follow the agent remotely across every site it visits. This can be done if the debugger supports a plugable remote interface, such as Java's JPDA [JPD] (Java Platform Debugger Architecture). One ULM debugger would follow the agent by connecting to each successive VM it visits to keep on debugging it.

In order to debug an agent, one would use the ULM disassembler, but it could be extended by adding information about the origins of each function and global variable. For example, it could be useful to add data to the OULM file format which would receive a history of migration data. It could map variables and functions to their origin site, module and line of code. A history of the agent's past migrations and travel itinerary could also be useful for the debugging of an agent.

Debugging a whole site on the other hand would require features such as the inspection of incoming and outgoing agents, the identification of loaded modules and their ubiquitous property, as well as the identification and list of ULM references.

We have talked with the author of Bugloo [Cia03], a source-level debugger for Bigloo and more specifically anything compiled to JVM bytecode. We have determined that it would not be hard to integrate Bugloo and our Bigloo VM (with Java backend) and Java VMs to efficiently debug ULM programs. At the same time we could use the various features Bugloo offers for the debugging of Bigloo FairThreads: the visual tracing of thread scheduling and signal emissions.

Currently the only support available in the ULM VMs for debugging includes a mere stack dump on error, with the location of the error in the source file provided by the `DEBUG` bytecodes inserted by the compiler.

9.2 Other enhancements

We have already talked about enhancements with regards to ubiquitous values, custom serialisers and missing module handling, but there are other enhancements we would like to see in the future.

9.2.1 A global garbage collector

Although we have not talked about it until now, there is a key piece of ULM which is still missing in the ULM implementation. We have already vaguely hinted at our custom GC, which cleans unused agent modules and signals on a per-site basis. The fact that we support references across sites is an entirely different problem: in order to determine that a reference can be disposed, one has to ascertain that each ULM site and agent in migration has stopped using that reference. This is called a *global garbage collector*: a GC which requires each ULM site to synchronise on the network to perform a garbage collection requiring data from each site at the same time. There are various ways to accomplish this, each with different features, side-effects and limitations.
The confection and fine-tuning of any GC is already a complex task [LQP92] [KMY94] [LC97], but we find that the implementation of a global GC is an even broader task which requires a high level of expertise. On the other hand, the lack of such a GC is only noticed after the creation of thousands of references by migrating agents, so it does not prevent us from testing ULM in many test cases. This is why we have postponed its implementation for later.

9.2.2 Mixin enhancements

There are several modifications which could be done on ULM mixins. For example, although there are various ways to execute some code during the mixin's instantiation, they are not satisfactory. In some cases we need a constructor function to be called during instantiation. Following the mindset of these mixins, we could add a new mixin clause such as `init` which would contain a block of code to execute during the mixin instantiation where the `super` and `this` variables are visible. There could be several such clauses, each would be executed in order of appearance once the `this` instance has been completed (all variables and methods added, removed or renamed and all inheritance done), and before it is returned by the `new` operator.
This construction would allow inheriters to change the content of inherited variables in a simpler way than presented in the Load Balancing example:

```
(define-mixin A
 (var foo 2))

(define-mixin B
 (inherit A)
 (init
  (set! this.foo 3)))

(define example (new B))
example.foo
=> 3
```

A better implementation for mixins would also be beneficial. Indeed right now mixins are implemented using various compile-time expansions and using hashtables to represent mixins at run-time. The use of a proper structure for mixins as well as appropriate support in the compiler would allow to add a lexical lookup for `this` and `super` fields which would allow us to remove most `this` and `super` qualifiers from mixin methods.

9.2.3 Miscellaneous enhancements

While we think the notion of *migration group* is essential, we find that the grouping could be done by other means than thread creation by agents. For example, it may be useful for an agent to create a thread outside its group[3]. It may also be desirable to dynamically link and unlink threads to agents.
For optimisation and efficiency purposes, it may be interesting to provide support in the VM (through new primitives for example) for functionality which would otherwise require the creation and migration of agents. For example, we could provide a way to emit signals on foreign sites without sending an agent there. Providing a simple interface for RPCs (Remote Procedure Calls) by spawning a thread on a remote site to invoke a function, rather than sending an agent and paying the cost of migration to do it.
Ubiquitous modules should be easier to write with an IDE. For example, when writing a ubiquitous module whose implementation can only be meaningful on one site (for example a phone's GUI) we should have a tool to generate the module's interface for the compilation site. Alternatively it should be possible to import certain modules by instructing the compiler that they are not available locally, but that certain variables are expected to come from them. Deployment of such modules should be automatised, for example by using a tool which sends and installs a module's implementation to a specified site and generates, sends and installs its interface to other specified sites. The addition of meaningful version numbers for modules could also be useful, especially in for agents which may visit several sites where ubiquitous modules may be out of synchronisation.

[3]It is currently only doable by creating an agent, which may incur an extra cost.

Chapter 10

Conclusion

On commence enfin à voir le bout du rouleau.

– Anonymous

For a moment, nothing happened. Then, after a second or so, nothing continued to happen.

– Douglas Adams

Throughout this dissertation, we have presented why and how we have implemented a new language for mobile agents. We have presented several problems and how we solve them with ULM. Finally we looked at what could be the future of ULM. At the end of this dissertation, we hope to have demonstrated that ULM presents new and valid solutions to the problems we have described.
Personally, after having used ULM for several years, I am convinced that the cooperative fair threads scheduling is a model that does not solve every problem induced by thread programming. Indeed it does solve what we see as the main problem of preemptive asynchronous scheduling: the non-deterministic scheduling. Finding a bug induced by a scheduling which is hardly reproducible is an impossible feat. The fact that ULM's scheduling is replayable helps a great deal in finding scheduling bugs.
Unfortunately our scheduling does not fix every problem, indeed it introduces a few specific ones. Our biggest gripe is that it is unavoidable that any large ULM library will feature cooperation points in hard-to-see locations. For instance, in a large program it is almost impossible to know which function call might cooperate. This poses the *instant skipping* problem: if one program expects to catch every emission of a signal (at every instant), and this program *skips* an instant unknowingly through a random library call, some signal emissions will be lost. Indeed we were confronted several times with skipped instants and signal loss. This is a problem very specific to our type of scheduling, but it is relatively easy to debug since the scheduling is replayable. On the other hand, instant skipping requires the use of mutexes as a necessary precaution in several places, which was one of the problems ULM's scheduling set out to fix.
The second problem we face with cooperative scheduling is modularity. Since agents can be written by different programmers, the injection of cooperation points in their program can be unevenly distributed. This means that some agents might cooperate every few seconds while others cooperate every minute. When these agents run at the same instants, this

will likely result in one agent executing its task before the other. It is in fact trivial in this model to abuse the scheduler to get more scheduling time than other agents. Indeed, even with the best intention of programming *fair* agents, the fact that agents come from different programs and programmers means their composition will never be fair.
After several years of playing with this fair cooperative scheduling I am convinced it is not less painful than asynchronous thread programming, but that it is rather a *different* kind of pain. I think this model of concurrency is suited for many different applications where asynchronoucy is not desired, but in as many other applications it may be preferable to use asynchronous threads. In any case it is a desirable model for those suited applications, which we believe there are many of. It would be interesting to work on lessening the problems induced by our thread model so that we could increase the usefulness of such models.
As for ULM's mobility features, I find that strong migration is much more intuitive than weak migration. Our component distribution application illustrates how easy it is to write agents for application installation as well as resource distribution. On the other hand, the use of a custom language to write mobile agents implies that either a lot of existing libraries have to be re-written for ULM, or that they have to be wrapped for ULM. While we have made every effort to ease the interfacing between ULM and other languages, we have also illustrated that it is never as easy as if the agent was written in the same language as the libraries which the agent wants to work on.
Contrary to our approach, there are many mobile agent systems which build on existing languages. By far the most popular one is Java, due to its relative *open* nature. Due to restrictions in the Java runtime it is not possible to implement strong migration without a custom runtime, but should such restrictions be lifted in a future version of Java (and there is so much work on mobile Java agents that it seems *plausible*), we think the tremendous popularity and library availability of Java would make it a better platform for mobile agents if we want them to spread outside of the labs.
Our work on ULM has produced a functional (in the sense of working) language distributed on our site at http://www.inria.fr/mimosa/Stephane.Epardaud/. We have invented a new model of fair thread scheduling, explored the implications of stack delegation and the unwinding of interleaved stacks, presented what we find a valid alternative to event loops, and explored a few realistic uses for agent mobility. We hope these techniques have been presented clearly and thoroughly enough to be used in other programs.

Bibliography

[23] SRFI 23. Error reporting mechanism. `http://srfi.schemers.org/srfi-23/srfi-23.html`.

[34] SRFI 34. Exception handling for programs. `http://srfi.schemers.org/srfi-34/srfi-34.html`.

[35] SRFI 35. Exception handling for programs. `http://srfi.schemers.org/srfi-35/srfi-35.html`.

[57] SRFI 57. Records. `http://srfi.schemers.org/srfi-57/`.

[9] SRFI 9. Defining record types. `http://srfi.schemers.org/srfi-9/srfi-9.html`.

[AAB+05] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. IBM Syst. J., 44(2):399–417, 2005.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, CBSE, volume 3054 of Lecture Notes in Computer Science, pages 7–22. Springer, 2004.

[BCS02] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In Proceedings of 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), Malaga (Spain), June 2002.

[BHV98] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. Jazz: an efficient compressed format for java archive files. In CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, page 7. IBM Press, 1998.

[Bou01] Gérard Boudol. The recursive record semantics of objects revisited. In Proceedings of the 10th European Symposium on Programming Languages and Systems, pages 269–283. Springer-Verlag, 2001.

[Bou03] Frédéric Boussinot. Concurrent programming with fair threads: The loft language. Technical report, 2003.

[Bou04a] G. Boudol. ULM: a core programming model for global computing. In Proceedings of ESOP 04, volume 2986 of Lecture Notes in Computer Science (LNCS), pages 234–248. Springer-Verlag Heidelberg, 2004.

[Bou04b] Frédéric Boussinot. Fairthreads: mixing cooperative and preemptive threads in c. Technical Report Research report 5039, INRIA, 2004.

[Bou04c] Frédéric Boussinot. Reactive programming of cellular automata. Technical Report Research report 5183, INRIA, 2004.

[Bre88] Thomas M. Breuel. Lexical closures for c++. In C++ Conference Proceedings, pages 293–304, Denver, CO, October 1988. USENIX.

[Cam] Objective Caml. Ocaml. `http://caml.inria.fr/ocaml/index.en.html`.

[Car94] Luca Cardelli. Obliq A language with distributed scope. Technical Report Research Report 122, Digital Equipment Corporation, System Research Center, Palo Alto, CA, March 1994.

[Car95] Luca Cardelli. A language with distributed scope. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 286–297. ACM Press, 1995.

[CF99] Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In ASAMA '99: Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents, page 22, Washington, DC, USA, 1999. IEEE Computer Society.

[CFLQ06] Giacomo Cabri, Luca Ferrari, Letizia Leonardi, and Raffaele Quitadamo. Strong agent mobility for aglets based on the ibm jikesrvm. In SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, pages 90–95, New York, NY, USA, 2006. ACM.

[Cia03] Damien Ciabrini. Bugloo: A source level debugger for scheme programs compiled into jvm bytecode. In Proceedings of the International Lisp Conference 2003, 2003. `http://www-sop.inria.fr/mimosa/fp/Bugloo`.

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

[Com] Apache Commons. Javaflow. `http://commons.apache.org/sandbox/javaflow/`.

[COR] CORRELATE. Brakes. `http://www.cs.kuleuven.ac.be/~eddy/BRAKES/brakes.html`.

[CSWD03] Jiannong Cao, Yudong Sun, Xianbin Wang, and Sajal K. Das. Scalable load balancing on distributed web servers using mobile agents. J. Parallel Distrib. Comput., 63(10):996–1005, 2003.

[ECM01] ECMA. C# Language Specification. Number 334. ECMA, 2001. `http://www.ecma-international.org`.

[Eng00] Ralf S. Engelschall. Portable multithreading. In USENIX Annual Technical Conference Proceedings, pages 239–250, San Diego, California, USA, June 2000. USENIX. http://www.gnu.org/software/pth/.

[Epa04] Stéphane Epardaud. Mobile reactive programming in ULM. In Olin Shivers and Oscar Waddell, editors, Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming, pages 87–98, Snowbird, Utah, September 22, 2004. Technical report TR600, Department of Computer Science, Indiana University. http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR60.

[FFF06] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In APLAS, pages 270–289, 2006.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 372–385, New York, NY, USA, 1996. ACM.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, pages 171–183, New York, NY, 1998.

[Fla05] Matthew Flatt. Inside PLT MzScheme. Rice University University of Utah, 300.3 edition, December 2005.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. IEEE Trans. Softw. Eng., 24(5):342–361, 1998.

[Gal] Erick Gallesio. Stklos. http://www.stklos.org/.

[Ger06] Guillaume Germain. Concurrency oriented programming in termite scheme. In ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang, pages 20–20, New York, NY, USA, 2006. ACM.

[GNU] GNU. Gnu compiler collection. http://gcc.gnu.org/.

[GS02] Erick Gallesio and Manuel Serrano. Biglook: a widget library for the scheme programming language. In 2002 Usenix annual technical conference, June 2002.

[IAN] IANA. http://www.iana.org.

[IEE95] IEEE. IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language). 1995.

[INR] MIMOSA INRIA. Fairthreads. http://www-sop.inria.fr/mimosa/rp/FairThreads.

[ISO99] ISO/IEC. ISO/IEC 9899 Programming languages – C. second edition edition, December 1999.

[JPD] The java platform debugger architecture. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. The Java Language Specification. Addison Wesley Professional, second edition, 2000.

[KCe98] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 Report on the Algorithmic Language Scheme. In Higher-Order and Symbolic Computation, volume 11 - 1, August 1998.

[KMY94] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, pages 79–88, New York, NY, USA, 1994. ACM.

[LC97] Sylvain R. Y. Louboutin and Vinny Cahill. Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97), page 516, Washington, DC, USA, 1997. IEEE Computer Society.

[LF05] Mario Latendresse and Marc Feeley. Generation of fast interpreters for huffman compressed bytecode. Sci. Comput. Program., 57(3):295–317, 2005.

[Lif] Second Life. Second life. http://www.secondlife.com.

[LOKK97] Danny B. Lange, Mitsuru Oshima, Günter Karjoth, and Kazuya Kosaka. Aglets: Programming mobile agents in java. In WWCA '97: Proceedings of the International Conference on Worldwide Computing and Its Applications, pages 253–266, London, UK, 1997. Springer-Verlag.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 39–50, New York, NY, USA, 1992. ACM.

[LY99] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison Wesley Professional, second edition, 1999.

[MGK03] Michael Sperber Martin Gasbichler, Eric Knauel and Richard A. Kelsey. How to add threads to a sequential language without getting tangled up. In Scheme Workshop 2003, November 2003.

[MMM02] A. Montresor, H. Meling, and A. Montresor. Messor: Load-balancing through a swarm of autonomous agents, 2002.

[MP05] Louis Mandel and Marc Pouzet. Reactiveml, a reactive extension to ml. In ACM International conference on Principles and Practice of Declarative Programming (PPDP'05), Lisbon, Portugal, July 2005.

[MW] Merriam-Webster. http://www.m-w.com.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes), January 1996.

[PCM+05] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, pages 216–227, New York, NY, USA, 2005. ACM.

[Pug99] William Pugh. Compressing java class files. In PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, pages 247–258, New York, NY, USA, 1999. ACM Press.

[Pyt] The Python programming language. `http://www.python.org`.

[Que96] Christian Queinnec. Lisp in Small Pieces. Cambridge University Press, 1996.

[RMH99] Derek Rayside, Evan Mamas, and Erik Hons. Compact java binaries for embedded systems. In CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, page 9. IBM Press, 1999.

[Rub] The Ruby programming language. `http://www.ruby-lang.org`.

[SBS04] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, pages 203–214, New York, NY, USA, 2004. ACM Press.

[SCD+] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Richard Kelsey, William Clinger, Jonathan Rees, Robert Bruce Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/`.

[SML] SML. Standard meta language of new jersey. `http://www.smlnj.org/`.

[SPDC06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A component model engineered with components and aspects. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, CBSE, volume 4063 of Lecture Notes in Computer Science, pages 139–153. Springer, 2006.

[SS02] Bernard Paul Serpette and Manuel Serrano. Compiling scheme to jvm bytecode:: a performance study. In ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pages 259–270, New York, NY, USA, 2002. ACM Press.

[SSCJ98] Onn Shehory, Katia Sycara, Prasad Chalasani, and Somesh Jha. Agent cloning: an approach to agent mobility and resource allocation. IEEE Communications Magazine, 36(7):58–67, 1998.

[Ste84] Guy L. Steele. COMMON LISP: the language. 1984. With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.

[Str00] Bjarne Stroustrup. The C++ Programming Language (Special 3rd Edition). Addison-Wesley Professional, February 2000.

[Sum00] Eijiro Sumii. An implementation of transparent migration on standard scheme. In Scheme and Functional Programming 2000, page 61, September 2000.

[Sus01] Jean-Ferdy Susini. L'approche réactive au dessus de Java: sémantique et implémentation des SugarCubes et de Junior. PhD thesis, Thèse de doctorat de l'ENSMP, September 2001.

[SW95] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In Static Analysis Symposium, pages 366–381, 1995.

[SY97] Tatsurou Sekiguchi and Akinori Yonezawa. A calculus with code mobility. In Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems, pages 21–36. Chapman & Hall, Ltd., 1997.

[TKT+07] Anand R. Tripathi, Devdatta Kulkarni, Harsha Talkad, Muralidhar Koka, Sandeep Karanth, Tanvir Ahmed, and Ivan Osipkov. Autonomic configuration and recovery in a mobile agent-based distributed event monitoring system: Research articles. Softw. Pract. Exper., 37(5):493–522, 2007.

[VWW96] Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in ERLANG (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[Wes03] Addison Wesley. The Unicode Standard: Version 4.0. Reading, Massashusetts, August 2003.

[Zim04] P. Zimmer. Récursion généralisée et inférence de types avec intersection. PhD thesis, Université de Nice Sophia Antipolis, France. INRIA Sophia Antipolis, 2004.

Appendix A

LURC

We have included an unpublished article written on a project related to ULM, named "LURC: Multiple flavours of thread with common semantics". It describes an implementation of a user-level thread library in C wich shares ULM's reactive primitives.

Abstract

It is often hard to choose the most suited threading model for an application amongst the many flavours of thread programming. Choices like preemptive versus cooperative, event-driven or reactive with signals usually drive one's adoption of a threading library. Yet sometimes, one model does not fit all threads in an application. LURC attempts to give the best of all models in a single unified scheduler, with semantics. It is a reactive cooperative multi-threading user-space C library, from which threads can become asynchronous when needed. LURC threads come in several flavours, each best suited for different needs, and all abiding by the same semantics, with the goals of being light, fast, deterministic (as in replayable) and customisable. In addition, event loop integration and a POSIX-like syntax make it easy to port existing programs to, and integrated syntax allows a clearer mixing of threading primitives into C syntax.

A.1 Introduction

When writing programs such as a Graphical User Interface, a client/server or an image processing library, a programmer has to face the fact that not only threads are often necessary, they are also complex. In a language such as C, which does not include threading support, there are many threading models, and even more threading libraries. Once a lot of time is spent in looking up the rough guides of each models and their libraries, the programmer is then faced with choosing the one that best fits its particular needs. Asynchronous or cooperative threads? Deterministic or non-deterministic scheduling? How do event loop models fit within each model? And then there's the question of efficiency and tuning to get the most out of your threads. Event then, when all this has been sorted out, the syntax of this threading model can be confusing: how does this fit in my language, why is this not part of the language?
Needless to say there is a certainly threading model appropriate for each situation, but finding one that is appropriate for every situation is a different matter. As we experience with several models we begin to recognise the benefits and drawbacks of each model, and

we also come to wonder what exactly is the real difference between those models, and their common ground? Processes have a higher context switch cost than threads in most cases, error-prone asynchronous threads easily show random crashes due to differently scheduled critical sections, while synchronous threads fail to benefit from multiple processors and event loops callbacks have no real stack state. At the same time all those models have features we want: we want multiple processor support, deterministic scheduling with semantics, perhaps even non-determinism (as long as it does not impede on the deterministic threads), and a way to integrate that pesky event loop that came with this or that library (try a GUI without one).
LURC is our attempt at providing several features found in different threading models, integrated in a way that makes sense. It is inspired by the FairThreads [INR] library, which provides cooperative threads in two flavours: threads with a stack (backed by an underlying POSIX [IEE95] Thread, aka Pthread), and automata threads with no stack. Their share a common deterministic scheduler, from which stateful threads can escape (and come back) and become fully asynchronous, thus benefiting from multiple processors or cores. Communication between threads is done via objects called signals (more on that later), which can be awaited and broadcast. ULM [Bou04a] is a programming model defining a set of primitives for reactive and mobile programming, whose reactive side extends FairThreads with new features such as expression-level suspension and weak preemption, as well as protection from preemption (think *finally*). LURC is based on our embedding of ULM in the Scheme programming language [Epa04], without mobility, but with asynchronisation and event loop integration.
LURC attempts to integrate several thread models in a C library, with an integrated syntax presenting the thread primitives as an extension to the C language, within the ISO [ISO99] C (or GCC [GNU] C for the syntax extensions) standard. LURC is designed as a lightweight library with customisable features, based on a highly optimised scheduler, introducing several novelties in reactive scheduling. It aims are being light and fast, adaptable to the user's need, whether on general purpose hardware, or on embedded platforms, while maintaining a deterministic (replayable) scheduling. The rest of this paper is organised as follows. We first present LURC's features in detail, whose implementation details are revealed next. We then present some related work, several benchmarks and future directions. Finally we conclude.

A.2 Lurc features

This section presents LURC's features.

A.2.1 Different types of threads

LURC supports four types of threads: two purely synchronous, the others able to switch from synchronous to asynchronous. The common goal for synchronous threads is to be executed by the Operating System within the same single native thread or process. This can be achieved in two ways in LURC: either the threads share a common stack, and each has to copy it back and forth in order to save (and restore) it when cooperating. Or each thread has its own separate stack, and the program jumps from one to the other when cooperating.
Copying the stack can be a costly operation for large stacks, but for small ones (threads

with few function calls on the stack), it is a very small price to pay. The main advantage is that since each synchronous thread is executed in the native process' stack, the system is responsible for growing it when needed, and each thread potentially has as much stack as any other process (which the OS does a good job at delaying allocation for). The other advantage is that for small stack usage, the space needed to save each thread's stack is very small, and thus a lot of threads can be created. These two advantages can easily be overlooked, but they are key to creating large numbers of threads: one of our tests involves the creation and scheduling of 200,000 threads with 150M of memory usage.
Having synchronous threads in a separate stack makes cooperation faster: there is no memory that needs copying. The downside is that the initial stack size has to be set to a sensible value. If every thread gets the same initial stack space as the native process, this limits the number of threads we can create, since these stacks won't be expendable by the OS, or benefit from any other optimisation on size the OS can do. It is possible however to use the `mmap` system call to resize an allocated stack, but on a 32-bit machine (which still accounts for the majority of CPUs) the memory address space is limited, and using mmap cannot change that. And if a thread stack cannot expand automatically and the thread runs in too small a stack, reaching the top will cause the program to crash at very unexpected places.
The last two types of threads start out as synchronous, but can be *detached* from the cooperative scheduler and run into asynchronous-land, until it decides to *reattach* to the scheduler and become synchronous again. These types of threads are not the default since we believe asynchronisation should be an explicit step from the programmer that decides to leave the safety of deterministic synchronous-land. We also believe that if programmers choose to use a deterministic thread system, they realise that the places in an application where threads **need** to be asynchronous are scarce. It comes at the cost of mapping each detachable thread to a native asynchronous thread, which implies a limitation on the number of detachable threads by the OS. The difference between these last two types of threads lies in the way they behave when they are synchronous. One type is always executed by its underlying Pthread, so its cooperation time (when relevant: i.e. when attached and cooperative) is equal to a native mutex a condition variable notification, and the underlying kernel context switch. The last type puts the underlying Pthread to sleep when synchronous and is executed in exactly the same way than purely cooperative threads with a separate stack, thus cooperation time is the same. For both asynchronous thread types, the memory usage for the stack is similar to that of the underlying native thread.
Each type of thread has its own advantage: minimal memory usage for the first type, fastest cooperation time for the second, and being able to become asynchronous for the last two, one of which has faster cooperation time when attached. It is worth noting that both synchronous thread types require no underlying Pthread, which can be handy in tight corners such as embedded systems. All types of threads behave exactly similarly with regards to the scheduling (when synchronous for the third type), so it is only a matter of tuning what's best for your application, or for each thread. With LURC these different types of threads are able to run next to one another, under strict adherence to common semantics. This provides a choice of implementation for each thread, while guaranteeing a strictly similar behaviour.
As you can see from this example, it is possible to create different types of threads, in a Pthread API fashion:

```
void thread_callback(void *args){
  // Do something useful.
}

void start_threads(void){
  lurc_thread_attr_t attr;

  // Initialise the attributes.
  lurc_thread_attr_init(&attr);

  // Ask for a shared stack thread.
  lurc_thread_attr_settype(&attr,  LURC_THREAD_SYNC_COPY_TYPE);
  // We want no pointer, no argument.
  lurc_thread_create(NULL, &attr, &thread_callback, NULL);

  // Now we want a detachable thread.
  lurc_thread_attr_settype(&attr, LURC_THREAD_ASYNC_LOCK_TYPE);
  // And start it.
  lurc_thread_create(NULL, &attr, &thread_callback, NULL);

  // Free the attributes.
  lurc_thread_attr_destroy(&attr);
}
```

In a similar way to the Pthread API, the `lurc_thread_attr_t` structure is used to hold information regarding the initialisation of LURC threads, which is later passed on to `lurc_thread_create` in order to create a new LURC thread. Here, after initialising this structure, we set the thread type to `LURC_THREAD_SYNC_COPY_TYPE` (purely cooperative with a shared stack), then call `lurc_thread_create` (the four arguments are: pointer to the new thread, initialisation attributes, function to execute in the new thread and argument to pass to that function) to create the new thread. We then proceed to create another thread of type `LURC_THREAD_ASYNC_LOCK_TYPE` (asynchronous with locks), and free the resources associated with `attr`. The two created threads will be started when the current thread cooperates.

A.2.2 Cooperative deterministic scheduling

Cooperative scheduling is achieved by dividing the execution time into abstract time slices called *instants*. During each instant each thread is allowed to react and can cooperate (with `lurc_pause`) with other threads by waiting for the next instant, or for a signal (more on that below). The scheduling in LURC is said to be *fair*, in the sense that every thread which wants to be executed (which is not waiting for a non-satisfied condition) is guaranteed to be executed at each instant. Fairness also implies that each thread has the same priority as every other, and ensures that there can be no thread *starvation*.
One of the greatest benefit of cooperative scheduling is that a deterministic behaviour can be enforced by semantics. LURC shares the reactive semantics of ULM, which makes every execution of a single program be scheduled in exactly the same predictable order. This is both very useful for program design, because you know exactly how the threads are going to be scheduled, and for debugging purposes, because a program cannot fail on random occasions due to scheduling. The same program is guaranteed by the semantics to be scheduled in exactly the same manner at every run: its execution is replayable (as far as scheduling is concerned).

The semantics of LURC is similar to ULM, with very few minor differences, and is thus not presented in this paper. Indeed we believe that we can view detaching threads as ULM agents who leave the scheduler for migration to a remote location, and attaching threads as ULM agents who join the scheduler at the beginning of a new instant. When LURC threads are detached, they can no longer use the cooperative API of LURC (cooperation or control blocks)[1], thus they can no longer interact with the cooperative threads (aside from the inherent nature of shared memory) and can be considered as if no longer there.
The scheduler is entered by calling `lurc_main`, which will then start all the created threads and schedule them until they all are done:

```
void printer_callback(void *args){
  char *name = (char*)args;
  int n = 4;

  while(n-- > 0){
    printf("Thread[%s]: %d\n", name, n);
    // Wait for the next instant.
    lurc_pause();
  }
}

void start_threads(void){
  // Two threads of default type.
  lurc_thread_create(NULL, NULL, &printer_callback, "A");
  lurc_thread_create(NULL, NULL, &printer_callback, "B");

  // Now schedule them and let them play.
  lurc_main();
}
```

A.2.3 Signals

Signals in LURC are objects which start every instant in an *non-emitted* state and can be changed to the *emitted* state exactly once during the instant. Threads can wait for non-emitted signals, which can mean waiting across instants and is a potential cooperation point (if the signal is non-emitted). Waiting for an already emitted signal simply returns immediately without cooperation. Signals can be shared by multiple threads and are thus visible by all. When a thread emits a signal (with `lurc_signal_emit`), all threads waiting for it (with `lurc_signal_await`) will be awoken and allowed to resume execution within the instant. The emitted state of a signal remains in effect until the next instant.
More complex *event* objects can be created using signals, in order to provide value association, single-thread notification or even multiple emissions in a single instant. It was the view of Gérard Boudol when creating ULM that the scheduler should concentrate on simple signals and leave the conception of more complex cooperation mechanisms we call events to the programmer. It is based on the idea that there are several orthogonal definitions of events, all of which can be implemented with signals, so it is a feature of the scheduler not to lock the programmer to a specific type of event, and to give him the means to create whatever event type fits his needs. In section A.5 we give an example of how this proved to be beneficial to us.

[1] This should however change in future versions of LURC as described in A.6.2.

In the following example, we use a signal to synchronise the two printers, so that the thread which prints "World" will always print it after the thread which prints "Hello". Doing so ensures the scheduling we want, despite the fact that starting the `world_printer` first would schedule it before the other.

```
void hello_printer(void *args){
  lurc_signal_t *relay = (lurc_signal_t*)args;
  while(1){
    printf("Hello ");
    // Emit the relay signal.
    lurc_signal_emit(relay);
    // Wait for the next instant.
    lurc_pause();
  }
}

void world_printer(void *args){
  lurc_signal_t *relay = (lurc_signal_t*)args;
  while(1){
    // Await the relay signal.
    lurc_signal_await(relay);
    printf("World\n");
    // Wait for the next instant.
    lurc_pause();
  }
}

void start_threads(void){
  // Create a signal.
  lurc_signal_t relay;
  lurc_signal_init(&relay, NULL);

  // Create two threads of default type.
  lurc_thread_create(NULL, NULL, &world_printer, &relay);
  lurc_thread_create(NULL, NULL, &hello_printer, &relay);

  // Now schedule them and let them play.
  lurc_main();
}
```

A.2.4 Integrated syntax

It is not possible in C to extend the syntax and therefore, not possible to introduce new statements for thread synchronisation. So in ISO C, if we want to execute a block of code in a special way regarding its scheduling (we call these blocks *control blocks*, we have to use a toplevel function for the block of code, and call a LURC library function by passing it a pointer to this function. Aside from destructuring the code (what if the **while** statement was a function call?), passing arguments to this toplevel function is always a problem.
In order to stay compatible with ISO C, LURC's control blocks are available in the form of library functions, taking pointers to functions and their arguments. But if LURC is used with the GCC compiler it offers a set of GCC-specific preprocessing macros which emulate a syntax extension to C, by providing inline control blocks, and by providing a way of declaring thread callbacks (their starting function) with automated packing and unpacking of arguments.

The following control blocks will be shown with this integrated syntax, in order to illustrate their usefulness. Remember that they are not necessary and are just a syntactic layer hiding the calls to the real library functions.
If the programmer still needs to use function pointers, LURC provides a set of ISO CPP macros named `LURC_CB*`, which declare callback functions with automatic packing and unpacking of arguments. For example, the `LURC_CB1(printer, char*, name){...}` declares a function which takes one argument of type `char*` named `name`, and several utility functions such as `printer_thread` or `printer_when`, which expand into calls to `lurc_thread_create` or `lurc_when` with the proper argument packing. This reduces greatly the amount of code the programmer has to write, and provides type-checking on callback arguments.

A.2.5 Control blocks

Control blocks are blocks of code whose behaviour depend on signals or scheduling. Since thread synchronisation is central to multi-thread programming, we view these blocks as though they were an extension to the C language, analogous to the **for** or **if** statements in C.

Suspension

A suspension block is a block of C code which can only be executed during the instants in which a given signal has been emitted. The block will be suspended at each beginning of instant until the given signal is emitted before it can resume execution. The library function is `lurc_when` and its syntactic extension is `LURC_WHEN`.
In the following example we have three threads: the first one prints its parameter string at each instant. The second prints it every instant when the `even` signal is emitted: it is suspended on this signal. The third thread emits the `even` signal every other instant, thus resulting in the first thread printing at every instant while the third one prints every other instant.

```
LURC_CB1(printer, char*, name){
  while(1){
    printf("Thread[%s]\n", name);
    lurc_pause();
  }
}

LURC_CB1(even, lurc_signal_t, sig){
  // Print only when sig is emitted.
  printer_when(&sig, "even");
}

LURC_CB1(even_emitter,
         lurc_signal_t, sig){
  while(1){
    // Even instant.
    lurc_signal_emit(&sig);
    lurc_pause();

    // Odd instant.
    lurc_pause();
  }
```

```
}

void start_threads(){
  lurc_signal_t sig;
  lurc_signal_init(&sig, NULL);

  // Start a printer at every instant. Notice that the two first NULL args
  // are the same as the two first lurc_thread_create arguments.
  printer_thread(NULL, NULL, "always");

  // Start a printer at every even instant.
  even_thread(NULL, NULL, sig);

  // And the emitter at even instants.
  even_emitter_thread(NULL, NULL, sig);
}
```

Weak preemption

Weak preemption is a means to control when and how a block of code can be preempted, akin to exception raising. A preemption block is a block of C code which can be halted (terminated, given up) when a given signal is emitted. This means that since signals are broadcast, threads can preempt themselves as well as other threads. However, preemption in LURC happens at the *End Of Instant* (EOI: period of time when the scheduler is terminating an instant and preparing to start the next one), and not right when the signal is emitted. This means that preempted preemption blocks will always resume their execution after the preemption block at the next instant. Weak preemption is available as the `lurc_watch` function or the `LURC_WATCH` syntactic extension.
Amongst other things, preemption blocks allow threads to get out of signal waiting, or suspension blocks. In the following example we define a function `exec_up_to_4` which is meant to be executed by a thread, and will execute the given callback function (`lurc_cb_t` is the type of callbacks in LURC) for at most 4 instants. This is done using preemption on a signal emitted by another threads in 4 instants.

```
// Print every instant.
LURC_CB0(printer){
  while(1){
    printf("Alive\n");
    lurc_pause();
  }
}

// Emits sig in n instants.
LURC_CB2(timer, lurc_signal_t*, sig,
                int, n){
  // Wait for n instants.
  while(n-- > 0){
   lurc_pause();
  }
  // Then emit the signal.
  lurc_signal_emit(sig);
}

// Executes cb for at most n instants.
```

```
void exec_up_to(lurc_cb_t cb, int n){
  // Create a signal for the preemption.
  lurc_signal_t sig;
  lurc_signal_init(&sig);

  // Emit sig in n instants.
  timer_thread(NULL, NULL, &sig, n);

  // Preempt cb when sig is emitted.
  LURC_WATCH(&sig){
    cb();
  }
}

// Now use all this.
void example(void){
  // Print "Alive" for 3 instants.
  exec_up_to(&printer, 3);
}
```

Protection

A protection block actually consists of two blocks of C code: a *protected* block, and a *protector* block. The effect is similar to **try/finally** blocks in C++ or Java. The protected block will be executed normally, but if preemption causes it to be halted, the next instant will start with execution of the protector block before propagating the preemption further up. If no preemption happens and the protected block terminates normally, the protector block is still executed. This makes sure that the protector block is always executed, whatever happens in the protected block. Protection blocks come in the forms of the `lurc_protect_with` function or the `LURC_PROTECT`/`LURC_WITH`/`LURC_END` syntactic macros. The next example illustrates the use for protector blocks by providing a `protected_read` function, used by threads wishing to read a file by blocks of 256 bytes each instant, while making sure that preemption will not cause the file descriptor to stay open.

```
void protected_read(char *file){
  FILE* f = fopen(file, "r");
  // Do some protected reading on f.
  LURC_PROTECT{
    // Do some reading on f.
    char buffer[256];
    int n;
    while((n = fread(buffer, sizeof(char), 256, f)) != 0){
      // Do something.
      printf("Read %d chars\n", n);
      // And cooperate.
      lurc_pause();
    }
  }LURC_WITH{
    // We must close this file.
    printf("Closing file\n");
    fclose(f);
  }LURC_END;
  // Here we are sure f has been closed.
}
```

A.2.6 Event loop integration

An event loop is a form of cooperative scheduling where callbacks are called atomically upon emission of an event by a single loop. The biggest limitation of event loops is callbacks cannot keep state across activations, thus seriously limiting the analogy with cooperative threads. Event loops on the other hand offer a way to transform most blocking function calls into an event, which does not block the event loop. Event loops are commonly found in client/server programs and graphical user interfaces, and often have to coexist with threading systems because they cannot be easily removed. How they coexist often proves to be problematic, even more so when the threading model is cooperative and both its scheduler and the event loop insist in being the *main loop*.
We have chosen to integrate an event loop in order to avoid most causes for detaching a thread: in FairThreads it is common to become asynchronous just to make a blocking function call, in order not to block the cooperative scheduler. We feel the required underlying native thread is wasted for this purpose and can easily be replaced by an event loop integration.
In order to integrate event loop based programs into cooperative programs, two special types of signals have been introduced: input/output signals and timeout signals. These signals are emitted automatically by the scheduler at the beginning of each instant when their specific condition have been met: either when a file description has available data to read (or data has been written), or when a given timeout has been reached
Because the event loop signals integrate well with the reactive signals approach, we call the part of reactive scheduling dealing with such signals the *Reactive Event Loop* (REL). REL signals are created like normal signals, but with a special type. There are four types of signals in LURC: plain, Input/Output notification, timeout or idle. The type of signal is specified upon signal initialisation via a `lurc_signal_attr_settype` function call.
The following example illustrates how the REL can be used to read asynchronous data, such as from sockets, by cooperating each time until a REL signal is emitted by the scheduler when data is available from the socket. As in the previous example, the function `protected_read` is meant to be called by a thread, and is preemption-safe.

```
void protected_read(int socket){
  // Prepare a signal for this socket.
  lurc_signal_attr_t attr;
  lurc_signal_t sig;

  // Initialise the attributes.
  lurc_signal_attr_init(&attr);
  // We want IO notification for read.
  lurc_signal_attr_settype(&attr, LURC_SIGNAL_IO_TYPE, socket, LURC_EVT_READ);
  lurc_signal_init(&sig, &attr);
  lurc_signal_attr_destroy(&attr);

  LURC_PROTECT{
    // Do some protected reading on socket.
    char buffer[256];
    int n;
    LURC_WHEN(&sig){
      while((n = read(socket, buffer, 256)) > 0){
        printf("Read %d chars\n", n);
        // Cooperate to reset the signal.
```

```
        lurc_pause();
      }
    }
  }LURC_WITH{
    // Close the socket and signal.
    printf("Closing down\n");
    close(socket);
    lurc_signal_destroy(sig);
  }LURC_END
  // Here we're sure socket has been closed.
}
```

In order to minimise the effort required to do common blocking IO operations, LURC provides a POSIX-like replacement API for several operations. Using this API is usually better since LURC REL signals can be more efficiently managed by LURC and it does not require a program to be rewritten to transform blocking IO calls into cooperation points (aside from renaming calls like *read* to *lurc_io_read* for example). Here is the same example as above but using LURC's POSIX replacement API.

```
void protected_read(int socket){
  LURC_PROTECT{
    // Do some protected reading on socket.
    char buffer[256];
    int n = lurc_io_read(socket, buffer, 256);
    printf("Read %d chars\n", n);
  }LURC_WITH{
    // Close the socket and signal.
    printf("Closing down\n");
    close(socket);
  }LURC_END
  // Here we're sure socket has been closed.
}
```

This approach has several limitations however. The first one is that calls to potentially blocking functions have to be replaced by their equivalent cooperating function in LURC. This can be simple using a preprocessor macro to replace all the function calls in your code. It can be trickier for third-party applications, through it is still possible to wrap the corresponding system calls at runtime. Future LURC versions will provide this option.
A second limitation is that obviously the corresponding cooperating functions have to be written, and possibly integrated in LURC. A number of them have already been defined, and we plan to add more in the future for most common blocking IO functions such as DNS calls (possibly using external libraries that support asynchronous calls to map them to our REL).
The inner workings of the REL is further discussed in the Implementation section.

A.2.7 Garbage Collector

LURC does not require a Garbage Collector, but if an application wishes to use LURC threads in combination with a GC (as is often the case when compiling a programming language that requires a GC to C), the GC has to know the threading system in order to collect the memory properly. This is usually done by extending the GC with support for each known threading library, requiring inside knowledge of the inner workings of the threading library in most cases, and thus being subject to implementation changes.

LURC supports GCs in general by providing an API for GCs to query LURC about where thread stacks are and how to stop and restart them. This should be enough for any GC implementation to work with LURC threads, since most GCs for C are limited to scanning the thread stacks without knowledge of what is in there. We tested our API in practise with the Boehm GC – probably the most used GC – by adding support for LURC, which proved to be fairly simple and efficient. We think any GC can be extended in the same way to support LURC threads.

A.2.8 Modularity

During the development of LURC we realised that each added feature greatly impacted with the others. Suspension and preemption are very simple on their own, but implementing the two features in a scheduler imposes a great deal of complexity, and performance is impacted, even when only one of them is used at a time. Protection suffers from the same problem, albeit less so, since protection without preemption makes no sense, and the combination of suspension, preemption and protection only adds a slight overhead compared to the previous two features.
A price is paid for each feature, even when it is not used, and we still want to have the fastest possible implementation of each of the features previously presented, while accepting the fact that some users will not have use for all the features. So we added the possibility to enable or disable each feature separately when compiling LURC: each of suspension, preemption, protection, detachable threads, the REL or the GC can be disabled and the compiled code will be optimised in circumstance, leading to the best scheduling of the wanted features. We feel it is very important if the LURC scheduler is to be used in constrained systems that the price paid for the scheduling fits the feature requirements.

A.3 Implementation

LURC's implementation is divided into two fairly independent parts: how threads are implemented, and how they are scheduled. The first part of this section describes the techniques used for creating the four types of threads LURC supports, and how context switches work between those different threads. The second part describes the techniques used to effectively minimise and distribute the scheduling algorithm between the context switches. On top of that, we will discuss in a third part the syntactic sugar offered by LURC to declare control blocks in C with no special preprocessing.

A.3.1 Threads

Having four kinds of threads aboard a single scheduler is no trivial task, but a necessary one since no single thread type could satisfy efficiently the three goals we strive for in our threads (speed, memory, detaching). Fortunately, both the thread creation and the context switching are easy to abstract and separate from the scheduling, allowing the addition of each new type of thread to have a very small impact on the scheduling code.

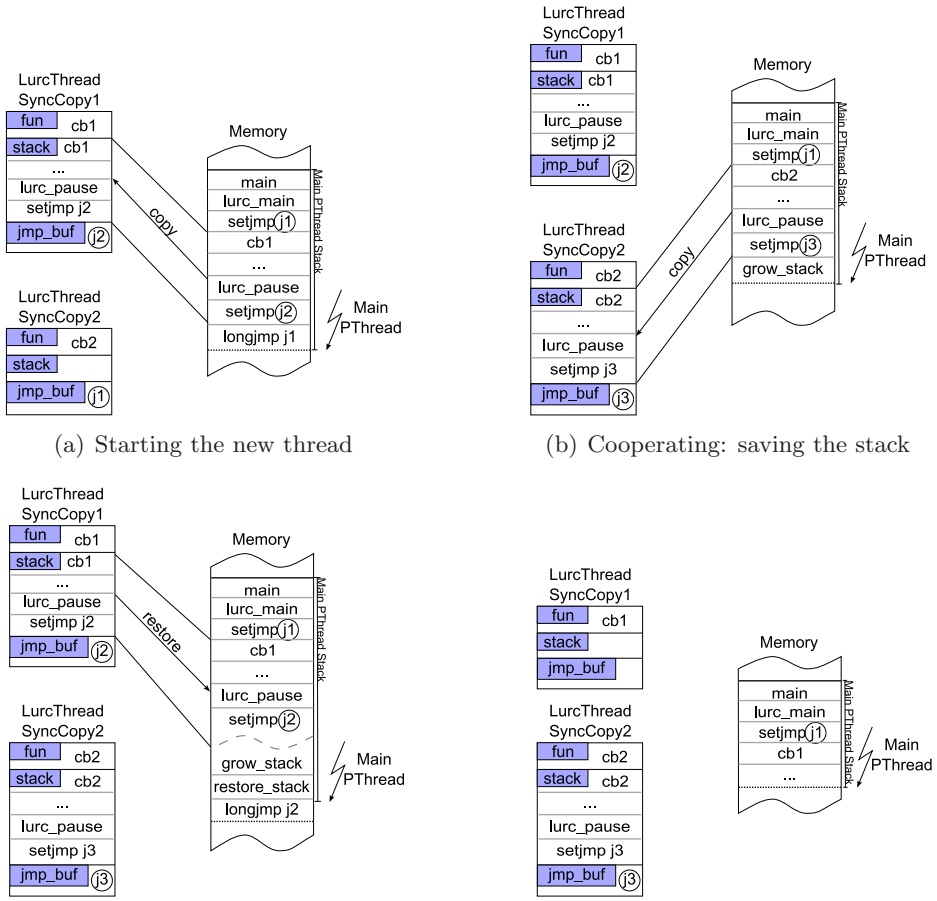
A note about native threads

We are going to describe how LURC threads are implemented. The first thing to know is that LURC threads are not kernel-level threads. They are user-level threads, and thus

rely on native (kernel-level) threads for execution. When there are no asynchronous LURC threads, the native process will be executing every LURC thread as described below. When there are asynchronous LURC threads, we rely on native threads in order to have asynchronous threads of execution with a shared memory. This is done with Pthreads, which is a portable threading API which gives us access to asynchronous threads on most platforms (and indeed there are very few platforms which provide only synchronous Pthreads). Since in Pthreads the native process (what a program runs in before starting any Pthread) is an implicit Pthread we call it the Main Pthread. In order to simplify the following section we call the native process the Main Pthread, regardless of whether or not there are any Pthreads in play.

A note about stacks

The stack is an area of memory in which a thread will be executed. Local variables and function parameters are allocated into it, and return addresses of functions are stored in it. Usually a stack is just a chunk of memory allocated in the heap somewhere. The initial and maximum sizes of a stack is set by the Operating System, but they usually are optimised such that only the used portion of the stack is effectively mapped in memory, and so that the minimum stack size is enough for most threads to avoid frequent remapping if the stack should grow. On Linux for example, the stack is allocated at the top of the heap, and grows downwards, while most heap allocation is done in the lower half of the heap. This means that the OS can grow the stack fairly without restraint if needed. Pthread stacks (except the Main Pthread, which because it is implicit, is at the top of the heap) are allocated in the middle of the heap, next to one another, which means that their maximum sizes are enforced. A Pthread stack usually is not lazily allocated, and can never grow past their maximum size since they would collide with one another.
The size considerations when talking about stacks are very important: what initial, effective and maximum sizes should be can play quite a large role in the performance of the thread. Ideally one would allocate an initial stack of a fair yet small size, and grow the allocated stack when needed, in order to keep the size and reallocation frequency low. But it is not possible to resize a stack: in C when you want to resize a chunk of memory, you use `realloc` which will try to increase the chunk's size. Unless it would collide with another allocated chunk, in which case `realloc` will move the chunk to an appropriate place in the heap and return the new location. This is not acceptable for a stack, since all variables allocated in the stack (local variables and function parameters) are accessed by address within this stack. So moving the stack around would require walking the stack to update all those addresses by the shift in stack location, which is clearly a complicated and costly feat in C. While there are in some OSes (like Linux) ways to circumvent this limitation, by mapping a chunk of memory to a certain virtual address (`mmap`) and being able to relocate the chunk while keeping the old mapped addresses valid (`mremap`), this is not portable and does not help so much because of memory space. Indeed if you allocate memory locations X to Y for a thread stack, and then memory locations Y to Z for the next stack, even if the first stack has been lazily allocated using `mmap` it cannot be remapped to a bigger size that would collide with the address space of the second thread. It is possible to prevent that by leaving enough space between each thread's stack's allocation, especially in a 64-bit address space. But in a 32-bit address space (which usually has reserved bits) with the default value on Linux for stacks (8M max) assuming we use all address space for thread stack allocation that still makes for a limit of 256 threads.



(a) Starting the new thread

(b) Cooperating: saving the stack

(c) Cooperating: restoring the stack

(d) Cooperation done

Figure A.1: Creating a Synchronous Copy Thread

The Synchronous Copy thread (shared stacks)

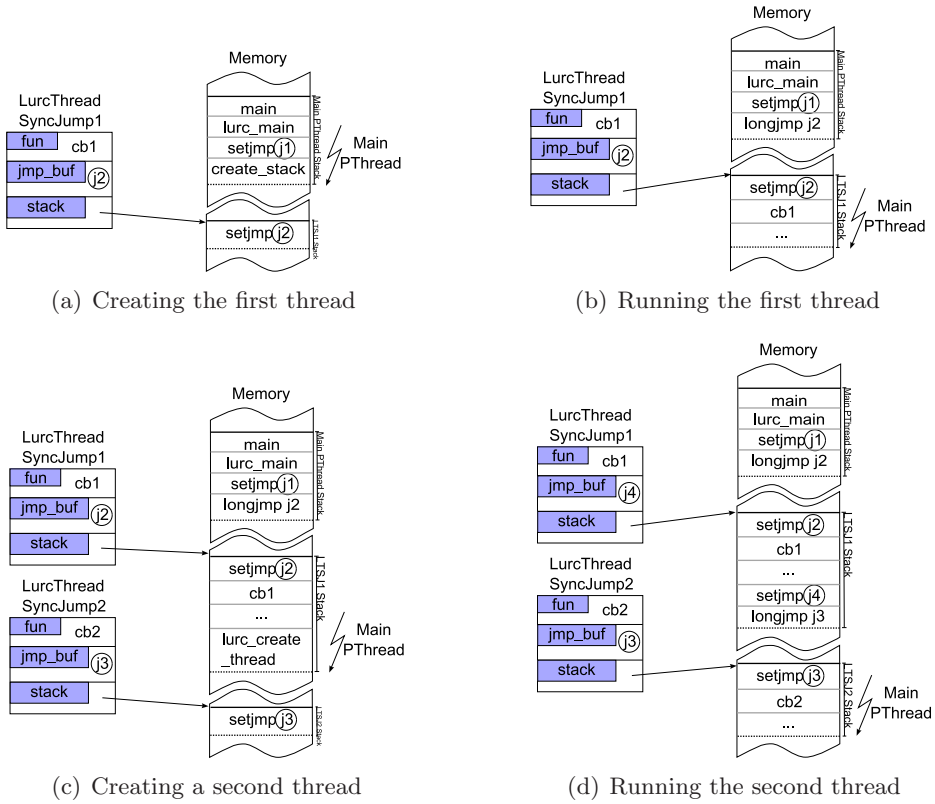
This is the default type of thread in LURC. Since only one synchronous thread is executed at a time, they are all executed by the same native process (that is, the process that called `lurc_main` described below). These threads run in the normal process' stack, and when not running the relevant portion of their stack is saved in the heap. Saving this thread's continuation (program counter and related CPU registers) is done using `setjmp`[2], and the switching to one of these thread done with `longjmp`. Its data structure consists of a `jmp_buf` (the thread's continuation), and a block of memory to save its stack when it is cooperating. The creation and cooperation of synchronous copy threads is illustrated in Figure A.1. It is divided in four steps *a* to *d*, each featuring to the left the data structures used to represent LURC threads, and to the right the native process' stack located in the heap. In these diagrams, the thunder-shaped arrow and label to the right of the stack indicates which native process is executing in the stack. Since we will further talk about Pthreads, we regard the native process as the main Pthread. The stack grows downwards, and thus the

[2]We do not use `sigsetjmp` merely because we handle signal masks elsewhere for implementation reasons.

`main` function is located at the top of the execution stack.
LURC scheduling is done within the `lurc_main` function. This function schedules every LURC thread until they all terminate, at which point `lurc_main` returns. Upon entering this function, a continuation is saved (named `j1` in the figure) which is used later on for starting new threads. Then the first created thread is started. In diagram *a*, the thread named LurcThreadSyncCopy1 is running. Its function `c1` has been called by `lurc_main` and it has at one point of its execution called `lurc_pause` in order to cooperate.
In order to save this thread's state we need to save its stack and its continuation. This is done by getting and saving the continuation `j2` and the stack from the call to `cb1` down to this continuation. Once this is saved, we can jump to the next thread's continuation. Here we have a brand new thread named LurcThreadSyncCopy2. New threads have an initial empty stack, and use the main continuation `j1` in order to start it. Once we jump to this continuation (we're back in `lurc_main`) we can start the new thread by calling its function `cb2` as shown in diagram *b*.
When this new thread wants to cooperate (back to the first thread in our example), it will do as seen before and save its continuation (`j3`) and stack, then jump to the next thread's continuation. But the next thread has a saved stack, so before jumping we need to restore it, and in that case it means growing the stack first to make sure there is enough space to restore the stack. Growing the stack is done simply by using recursion to allocate stack memory until there is enough space to restore the stack without overwriting over the current function.
In diagram *c* the function `grow_stack` has grown the stack enough and copies the first thread's stack over its upper stack, then jumps to the first thread's continuation to resume execution as shown in diagram *d*.
Of course, sometimes cooperation occurs between a large stack to a small stack and there is no need to grow it, but cooperation always costs two memory copying operations: the saving and the restoring. This technique is very old, and still used today by many threading implementations (such as MzScheme Virtual Machine [Fla05]).
While this allows for a very large number of threads because the memory usage of each thread is strictly limited to its stack usage and thread creation is not limited by address space, it also means that all these threads share a common stack address space and cannot communicate stack pointers with one another. We think it is a small price to pay when one needs to have hundreds of thousands of threads running with minimal memory usage.

The Synchronous Jump thread (separate stacks)

This type of cooperative thread is executed in its own stack space, which thus has no need to be saved and restored. Like the first type of LURC thread, it is executed by the main process and its continuation is saved by `setjmp`. Its data structure consists of a `jmp_buf` (the continuation) and a block of memory for its stack. This block of memory is allocated when creating the thread and spans its lifetime. The downside is that it cannot be resized (since that would mean it could be relocated and stack variables' addresses would become invalid).
The technique for creating this type of thread has been illustrated by GNU/Pth [Eng00], by using an alternate signal handler stack to bootstrap a block of memory into an executable stack. The problem they solved was how to turn a block of memory into a stack we can jump to, to start executing the thread's code. The idea is that the current process sends itself a signal, asking for the signal handler to be called on that new block of memory.



(a) Creating the first thread

(b) Running the first thread

(c) Creating a second thread

(d) Running the second thread

Figure A.2: Creating a Synchronous Jump Thread

The signal handler then saves the continuation and returns. Upon completion of the signal handler code, the saved continuation can be used by the new thread to jump into its new stack.

In diagram *c* we create a new thread of the same type, thus allocating a new stack for it and getting an initial continuation of `j3`. Cooperating from the first thread to the second is shown in diagram *d*. It consists simply in saving the continuation (`j4`) and jumping to the new thread to start its function `cb2`. No memory copy operation is required when cooperating to and from this type of thread.

The Asynchronous Locked threads

This is a type of thread that can become asynchronous, but starts out as synchronous Since a thread's stack cannot be relocated to a different address, if a thread starts executing in one stack, this stack's address can never change throughout its execution. The easiest and most portable way to have a thread asynchronous is by using Pthreads as the underlying thread. Pthreads also cannot have their stacks moved, thus if a thread is to be made asynchronous at any time of its execution, it must be mapped to a Pthread during the whole time, even while cooperative.

So this type of thread is always executed in a Pthread, and Pthread mutex locks and condition variables are used to make sure that only one cooperative thread is executed at

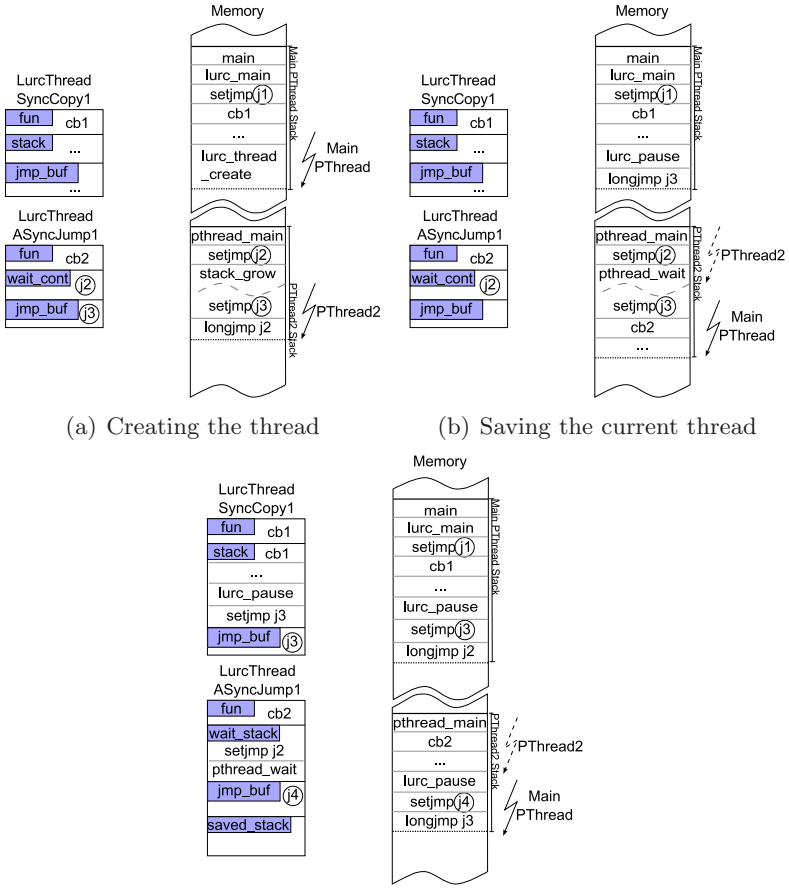
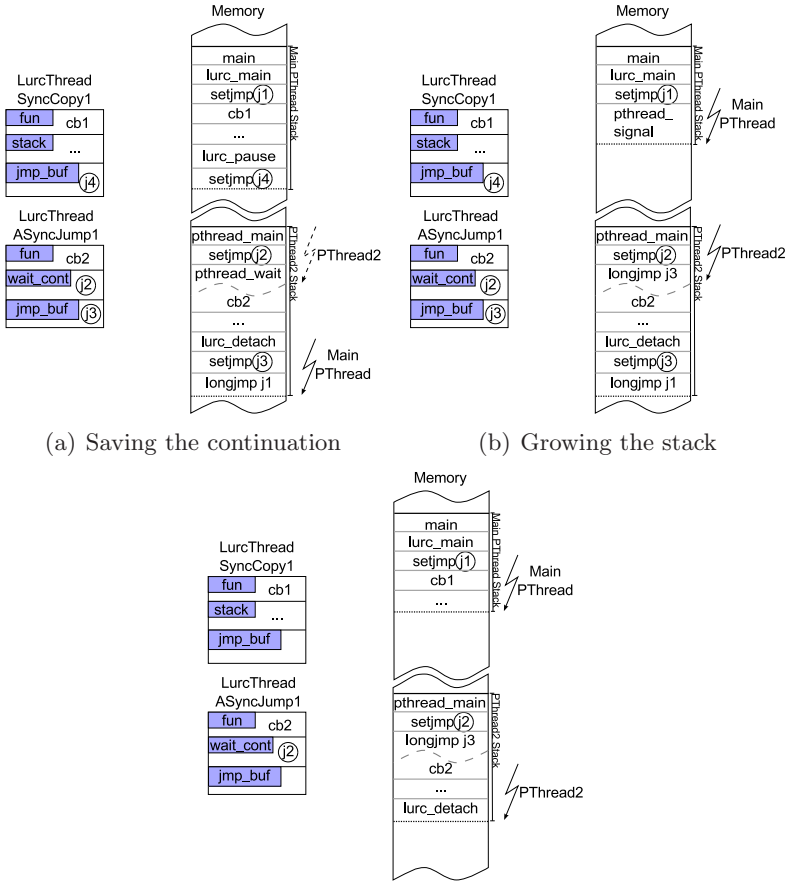


Figure A.3: Creating an Asynchronous Jump Thread

the same time. Since in Pthreads the main process is also a Pthread, you can think of all purely cooperative threads (the first two types) as executing in the main Pthread, while detachable LURC threads execute each in their own Pthread.
Cooperating in and out of this type of thread is done using locks, and the continuation does not need to be saved, nor does the stack. Becoming asynchronous is done simply by removing the thread from the scheduler and cooperating to the next thread without waiting for it to cooperate, and simply keep on executing asynchronously. Attaching on the other hand requires grabbing a mutex to insert the thread into a list of threads to attach, to be incorporated in the scheduler at the next instant.

Asynchronous Jump threads

This type of thread can also become asynchronous, but it differs with the *lock* type when it is cooperative. The main idea is that for threads which are mainly cooperative and only become asynchronous very few times, we want to improve the cooperation time. Like the previous type of thread, we need a Pthread underneath the LURC thread, but instead of doing the synchronous cooperation with Pthread locks, we put the Pthread in a stasis,



(a) Saving the continuation

(b) Growing the stack

(c) Restoring the continuation

Figure A.4: Detaching an Asynchronous Jump Thread

while the main Pthread (the one that executes both purely cooperative types of threads) goes in his stack to execute the thread, just like the Synchronous Jump type of thread.
Figure A.3 illustrates the process of creating such a thread. In diagram *a*, we have a Synchronous Copy thread named `SyncCopy1` which creates a thread of type Asynchronous Jump named `ASyncJump1`. We do that by creating a Pthread in which we save a top continuation we call the *waiting continuation*, then we grow the stack by using recursion and stack allocation, in order to leave a fixed amount of space. This space will later be used to put the Pthread in a stasis. After having left that space, we save a *working continuation* `j3` which will later be used as the entry point for the main Pthread, then the thread jumps to the waiting continuation `j2` in order to leave the *working stack* for the *waiting stack*. It then enters the stasis by awaiting a Pthread condition variable. This thread is now ready to be started when the current thread cooperates.
Its first activation is shown in diagram *b*, when the current thread cooperates to it by jumping in its working continuation, and starting its entry function. From this time on, the underlying Pthread will remain waiting in a stasis, while the main Pthread will jump in and out of its working stack just as it does for the Synchronous Jump threads.
In Figure A.4 we present how detaching is done. In diagram *a* we start with our pre-

vious scenario where the `SyncCopy1` thread cooperated to our `ASyncJump1` thread. The latter thread now wants to detach from the scheduler and become asynchronous. It calls `lurc_detach`, which saves the working continuation `j3`, then jumps in the main continuation `j1` in order to get away from the `ASyncJump1` stack. In diagram *b* it then signals `Pthread2` in order to awaken it from its stasis. From then on, the main Pthread cooperates to the next synchronous thread, and the `AsyncJump1` thread is asynchronous: after awakening from the stasis it jumps back to the working continuation `j3` and can thus return from the `lurc_detach` call in diagram *c*.
This is an interesting type of thread because it can be executed by two underlying native threads in its lifetime. If we compare it to the Asynchronous Lock type, cooperation is faster when attached, while attaching and detaching is probably a little more expensive.

A.3.2 Scheduling

What can we optimise ?

One of the goals of LURC is, of course, to have an efficient scheduling. Traditional FairThread-like scheduling (of threads, not automata), involves two main scheduling places: at the end of instant, and when emitting a signal. In only these two places is determined the list of threads allowed to run, which means that cooperation consists simply in cooperating to the next thread in the list (constant time), or entering the end of instant if there is none. The end of instant is the phase where a lot of things happen: preempted threads get actually preempted, attaching threads are inserted, signals are reset, REL signals are emitted and threads are traversed in order to determine whether they will be scheduled or not. This means that the time spent in this phase is proportional to the number of threads to preempt, insert, and inspect for scheduling, as well as for signal to reset and emit.
Signals to reset is a phase already made costless in previous FairThread optimisations by associating the *emitted* state of the signal with the instant number in which it was emitted. Thus increasing the instant number automatically resets every signal.
Emitting signals from the REL cannot be optimised, since only the REL knows which signals will be emitted, the only place we can make this faster is by emitting them all at once instead of sequentially, at the cost of sanity in the **emit** function: we will see later that scheduling takes mostly place there, making it a complex function. Constructing the set of objects used during the REL evaluation (in our case, arguments to **select**) can be cached efficiently and reconstructed only when modified.
Inserting attaching threads requires a mutex to make sure the list does not change while incorporating them. Other than that it comes down to appending this list to the list of threads to run (since detached threads cannot have a suspension/preemption context and cannot be waiting for a signal, they must be runnable), which is constant.
So the only things left to optimise are preemption and inspection for runnable state. These two things can be done at once, but it still means the traversal is proportional to the number of threads inspected, which results in the end of instant having a cost of O(t+s) with t the number of threads to inspect (all but those detached) and s the number of REL signals to emit.

Introducing the End Of Action phase

As stated previously, scheduling happens primarily in the EOI phase, and when emitting a signal. When emitting a signal we have to determine which threads are going to be

scheduled later within the instant as a result of the signal emission. On the other hand, during the End Of Instant phase, we have to determine which threads are going to be scheduled during the next instant.

Since the time spent in the EOI is proportional to the number of threads to examine, the more you have threads, the more time the scheduler is going to take at the EOI. This means that there will be a lag at the EOI when you have many threads running. What we propose is to spread as much of the EOI scheduling as possible during the instant, by introducing the concept of an End Of Action phase.

Suppose a thread wants to cooperate with `lurc_pause`, which means it will be waiting for the next instant. The semantics of `lurc_pause` impose that the thread will not do anything anymore during the current instant: it will not do a thing until the next instant, regardless of whether preemption will apply at the EOI, or whether suspension clauses will cause it not to be scheduled during the next instant. When a thread is done for an instant, we call this time the EOA for this thread, and we try to schedule it for the next instant before the EOI.

Indeed, for several situations we do not need to await the EOI to know what this thread will be doing in the next instant. Suppose it does not have any preemption or suspension context: at the EOI the scheduler will determine that the thread has to run at the next instant. Guess what: we know it already! So we have a list of threads to schedule at the next instant for sure, and threads reaching their EOA without any constraints (preemption or suspension) will just append themselves in this list, which will be used as a starting point when building the list of threads to run at the EOI, thus removing the need to examine this thread at the EOI.

Now what happens at the EOA for a thread with preemption? Well, once again there are some things we are certain of: if the preemption signal has already been emitted, the thread will be preempted for sure (whether or not the protector clauses will be allowed to run at the next instant due to suspension) and so we can already prepare the preemption to see whether some protector clauses will have to be run or not, and what context (w.r.t. suspension) this thread will have at the next instant. If this thread has no suspension context for the next instant, we also put it in the list of thread to run at the next instant.

The other possibility when looking at a thread at the EOA with a preemption context is when the preemption signal has not yet been emitted. We have no way of knowing whether the signal will be emitted or not during the instant later on. So since we don't know, we'll put it in the list of threads to run at the next instant too, and if that signal is emitted later on, we'll reconsider that choice. A thread for which we planned preemption can also be re-preempted later on during the instant by a higher-level preemption clause. So when emitting a signal which triggers preemption for a thread which has passed its EOA, we can reconsider the preemption for the next instant.

An EOA for a thread which has a suspension context can be simple too: at the next instant it will have to wait for its topmost suspension clause's signal. Since the thread was running until its EOA it means all its suspension clauses were satisfied, all those signals were emitted, and so the list of threads waiting for them in the signal objects must be empty, and will remain empty until the EOI when we determine which thread will have to wait for which signals, and be put in the waiting lists. But wait, if this list is to remain empty until the EOI, why don't we fill it now? We use the waiting list of a signal in two ways depending on whether the signal has been emitted or not: if the signal has not been emitted yet (his emission counter is inferior to the current instant), the list consists of the

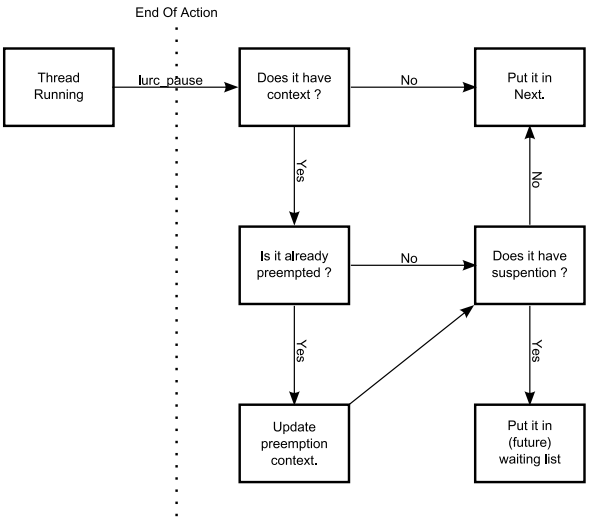


Figure A.5: The End Of Action

threads waiting for the signal for this instant. If the signal has already been emitted, it will not be emitted anymore during the instant, and so the list holds the threads waiting for this signal, but only after the EOI, when the instant counter will be increased and automagically shift this list from the second type of use to the first type.
In practise, we can already schedule threads with a suspension context for the next instant when there are no preemption clauses above the topmost suspension clause. If there are preemption clauses below and they are triggered between the EOA and the EOI, we simply mark the thread so that it knows where to continue when it will be allowed to, but this does not change its scheduling for the next instant. Now if there are preemption clauses above the topmost suspension clause, either the preemption is assured (the signal has already been emitted) and the thread is treated as a thread with preemption and no suspension (the suspension clause would be preempted at the EOI anyways). And if the preemption is not assured we assume it will be waiting for the suspension signal at the next instant, and rely on the emission of the preemption signal to reconsider that case and cancel the suspension.
The End Of Action phase is summarized in Figure A.5 .

The not-EOA cases

There are two more complex cases of scheduling before the EOI, the first one is for threads which do not wait for the next instant, but wait for an absent signal. This is not a case of EOA, since the thread might be awoken later during the instant. But there are only two ways threads can reach the next instant: waiting for it with `lurc_pause` which we already covered, or by waiting for a signal not emitted during an instant. So this is the only case left. What we do is assume the signal will never be emitted and treat the thread as if it reached its EOA as described above. If the signal is emitted during the instant, we simply cancel its scheduling for the next instant.
The second complex case is how to schedule threads which do not even get executed during

the instant, because they are waiting for an absent signal. The previous cases considered the next instant scheduling at the EOA, but some threads do not have any of the *action* in EOA. This is always threads waiting for a signal which does not get emitted, but we usually don't have to prepare their next instant's scheduling, since they will still be awaiting the same signal if it does not get emitted during this instant. In fact we don't even look at them if we don't wake them up, or preempt them. But suppose a signal is emitted which would preempt the thread's waiting at the EOI: we are now looking at a thread exactly in the first non-EOA case, and plan its next instant in the same way.

Unschedulable things left to do at the EOI

With the techniques presented here there is virtually nothing left at the EOI: all the scheduling is done during signal emission or at the EOA. But there is one exception, which is twisted enough not to happen often: when a thread is waiting for a signal, which will not be emitted during the instant, and this waiting will be preempted at the EOI, but there is a suspension clause above the preemption clause. Here is such an example:
This is far fetched and in our experience seldom happens, but we have no way to treat this: at the EOI the thread will still be waiting for the bottom signal (the preempted wait), but within the instant we already know it will not be waiting for it at the next instant, but instead will be waiting for the top signal (the one above the preemption clause). We cannot remove the thread from the bottom signal's waiting list until the EOI since we don't know it will not be emitted, so at the EOI we have to remove it from this list and insert it in the top signal's waiting list. For that we have created a list of threads to visit at the EOI, in which we put all threads that meet this criteria when emitting a preemption signal.
Aside from this rare case, the only things left to do at the EOI are: treat the special cases, increment the instant number, treat the REL and incorporate the attaching threads. To the best of our knowledge, the only things we could further remove from the EOI are the REL and attaching threads, if we change the semantics and consider that external signals can be emitted and threads can reattach during the instant, at the cost of increasing the number of times we must call `select` and grab locks, which we do not think would be of any benefit.

EOA scheduling: the conclusion

What our scheduler achieves in terms of scheduling is a minimal load at the EOI, by spreading the scheduling as much as possible within the instant, and while this does not reduce the complexity, it does spread it as much and as fairly as possible, in order to reduce the EOI time to a minimum, so as to hide the impact of the scheduler's implementation from a program.

A.3.3 Syntactic sugar

Control blocks and macros

When introducing threads and control blocks in C, perhaps the most striking and painful thing for the user is the syntax problem: there are no anonymous functions in C and so every library function dealing with code control has to deal with function pointers. For each new thread you start you have to declare a new function for its body, figure a way to pass it parameters (no variable capture there to help you), and give pointers to this

function when creating the thread. The same goes for control blocks: LURC is a C library and as such, does not introduce new blocks similar to the `if` statement in C: you always have to declare a function with the control block's body, and pass a pointer to this function to LURC in order to enter a new control block with this function as body.
This is both tedious, very frustrating to users, and makes both reading and writing code cryptic at best. In ISO C there is no way to add new instructions to the language without changing the compiler, but there is a way to use some macros to transform new instructions into ISO C. However, using ISO macros there is no way to introduce control blocks in a function: the only workaround would be to introduce the code needed to setup the control block above it and the cleanup code below it, but that would expose some of the LURC library internals and lead to a lot of code duplication.
What we really want is to transform this macro into something calling the equivalent library function. This is not directly possible since blocks of code in C are not reified. But where ISO C has shortcomings, GCC C offers new solutions. GCC has long had a set of extensions to ISO C, which tend to be at the same time useful, widely used, and very often end up in the next version of ISO C. Moreover, GCC is one of the most widely used C compiler, and available in a wide range of Operating Systems. One of the particular extension of GCC C is that of nested functions [Bre88], which allow functions to be defined inside other functions, thus giving a name to a block of code, which will on top of that capture the current environment and allow the nested function to use any variable declared in the outside function. The only limitation is the lifespan of the inner function which ends when the outside function returns, but this is good enough for us. The `LURC_WHEN` macro can thus be implemented as follows:

```
// Defining it thus.
#define LURC_WHEN(sig)                        \
 auto void __gensym_function(void*);          \
 lurc_when(sig, &__gensym_function, NULL);    \
 auto void __gensym_function(void* __gensym_p)

// Allows us to call it thus.
LURC_WHEN(&signal){
  printf("yeepee\n");
}

// Which expands in.
auto void __lurc_l112_function(void*);
lurc_when(sig, &__lurc_l112_function, NULL);
auto void __lurc_l112_function(void* __lurc_l112_p){
  printf("yeepee\n");
}
```

The `auto` modifier for the function is required for nested functions, and declaring it before defining it allows us to use it in the call to `lurc_when` before it is defined by leaving the control block which follows the macro call. The only name pollution introduced in user code is the name of the nested function and its parameter, for which we use a crude gensym macro to generate a unique name based on the line number, which is the best we could do with C macros. The only restriction is that there is only one `LURC_WHEN` per line, but we prefer this limitation which will be detected at compilation by an error message (though cryptic) rather than have an obtrusive syntax for our control blocks.

However this works for the suspension and preemption blocks, but the protection block is more complex since it requires two nested functions, via two macro calls. Due to the gensym generated names for those functions, the first macro call will not be able to declare the second nested function since it does not know its name (line number), and the other way around: the second macro call will not be able to call the first nested function because it does not know its name (line again). So we are forced to use static names for both, and in order to be able to use several protection blocks in a single C function, we put the whole block in a nested C block, thus allowing us to both override previously declared nested functions, and restricting their scope downwards. But this requires an extra third macro call after the second block in order to close the block. The following explains how we did it:

```
// Defining it thus.
#define LURC_PROTECT { \
 auto void __lurc_fprotected_function(void*)

#define LURC_WITH \
 auto void __lurc_fprotector_function(void*)

#define LURC_END lurc_protect_with( &__lurc_fprotected_function, NULL,  \
                                    &__lurc_fprotector_function, NULL); \
 }

// Allows us to call it thus.
LURC_PROTECT{
  printf("yeepee\n");
}LURC_WITH{
  printf("ouch\n");
}LURC_END

// Which expands in.
{
 auto void __lurc_fprotected_function(void*){
   printf("yeepee\n");
 }
 auto void __lurc_fprotector_function(void*){
   printf("ouch\n");
 }
 lurc_protect_with(&__lurc_fprotected_function, NULL,
                   &__lurc_fprotector_function, NULL);
}
```

While this is not exactly what we want (it requires an end tag) it is a decent compromise because it allows us to use the normal (non syntactic) LURC API. Getting rid of the end tag would require the injection of internal LURC code as part of the `LURC_PROTECT` and `LURC_WITH` expansion, which we do not want.
The last place we'd want syntactic sugar is for creating threads. Ideally we could do the following:

```
void start_threads(int n){
  if(n > 0){
    LURC_THREAD{
      printf("Thread #%d\n", n);
    }
```

```
    start_threads(n - 1);
  }
}
```

We have decided against it for several reasons. First, this syntax requires environment capture, so we could use the same trick as before and use macros expanding it thus:

```
void start_threads(int n){
  if(n > 0){
    auto void __lurc_l143_function(void*);
    lurc_thread_create(NULL, NULL, &__lurc_l143_function, NULL);
    auto void __lurc_l143_function(void* __lurc_l143_p){
      printf("Thread #%d\n", n);
    }
    start_threads(n - 1);
  }
}
```

But we're faced with one problem: creating a thread does not start it, and therefore, we may (and in this case will) exit from the `start_threads` function before starting all these new threads. Nested functions however, require that their containing function does not return in order to work, so this would not work. The only way to make it work would be to save the current stack when creating the threads, and so start these threads with an initial stack containing the captured environment, and possibly much more. This means several things: bigger memory impact, since these threads would have an initial stack remaining throughout their entire lifespan, whether or not the captured environment is used. It also means that the captured environment is not shared but duplicated, so if any of the creator or created thread modifies a local variable, the other will not see it, thus differing from the control blocks semantics where this would work. And last but not least, this would not allow us to create detachable threads, since they require starting an underlying Pthread, which cannot start with an initial stack. For all these reasons we have chosen to not allow inline thread blocks and require each thread to start with a clean stack via the library function call.
We should note that since these macros use GCC C features, they are only enabled when using the GCC compiler. Moreover, as GCC's nested functions sometimes require an executable stack, and some operating systems (or specific distributions of them) explicitly disallow that, on these systems too our macros are not enabled. This is not too much of a problem however since these macros merely use the standard LURC API to make the code simpler. On those systems, the user will have to stick to the standard LURC API.

Passing arguments to callbacks

When we need to pass a pointer to some code in C, we usually use a function pointer, and an associated `void*` argument in order to be able to pass it some data. This is the case in LURC too, since LURC functions such as `lurc_thread` have to take a function pointer of a specified type, and we cannot write a `lurc_thread` function for every type of function pointer that could be passed to it. So limiting the prototype of this function to `void f(void*)` is a good compromise, since all the possible needed arguments can be packed in a structure whose pointer would be the `void*` argument.
Of course this is cumbersome as each function needs a structure definition for its packed arguments, to unpack its arguments and each call to functions such as `lurc_thread` needs

to pack them. On top of that it is very easy to pass the wrong structure as argument since no checking is done at compile-time on argument types.
We have already shown in Section A.2.5 how we use macros such as `LURC_CB1` to declare a callback that takes more than one argument and specifies their types. On top of that, this macro generates several functions that call LURC functions such as `lurc_thread` or `lurc_when` which take the correct number and types of arguments. This is possible with the use of simple ISO C macros, with the slight annoyance that argument names and their types have to be separated by commas, and their number has to be specified in the macro name. We believe however these to be minor when compared to the benefits they offer.

Just syntactic sugar?

Here is an example of a function combining an array of signals into one using the *or* combiner:

```
// A callback taking 3 arguments.
LURC_CB3(or_combiner,
         lurc_signal_t*, kill,
         lurc_signal_t*, ok,
         lurc_signal_t*, sig){
  LURC_WATCH(kill){
    while(1){
      lurc_await(sig);
      lurc_signal_emit(ok);
      lurc_pause();
    }
  }
}

lurc_signal_t *
combine_or(lurc_signal_t *kill, int n, lurc_signal_t **sigs){
  lurc_signal_t *ok = lurc_signal("ok");
  for(int i=0 ; i < n ; i++){
    // This function is generated for the or_combiner callback up there.
    or_combiner_thread(NULL, NULL, kill, ok, sigs[i]);
  }
  return ok;
}

void
when_or(int n, lurc_signal_t **sigs, lurc_cb_t cb, void *param){
  lurc_signal_t *kill = lurc_signal("kill");
  lurc_signal_t *ok = combine_or(kill, n, sigs);

  LURC_PROTECT{
    LURC_WHEN(ok){
      cb(param);
    }
  }LURC_WITH{
    lurc_signal_emit(kill);
    lurc_signal_destroy(kill);
    lurc_signal_destroy(ok);
  }LURC_END
}
```

Combining a set of signals with *or* requires to launch one thread per signal, each waiting continuously for one of the signal, and emitting the `ok` signal when their signal is emitted. We need a `kill` signal to stop all those threads when the `ok` signal is not used anymore, so each thread is preempted by this signal. Having this combined signal allows us to do a `when_or` function, which takes a set of signals and a callback, which it calls in a suspension context on any of the signals. The protection block is used to stop the created threads and deallocate the required signals when exiting from the callback.
This example illustrates all the syntactic sugar available in LURC. We think that this code is clear to read, and intuitive to write, mostly thanks to the syntactic sugar, without which the code would be much longer and verbose. In the end we feel that when it comes to using a thread/scheduler system, syntax plays a large part in its acceptance. If code is hard to write or/and read, the system will not be used, which is why we spent some effort into this. But for those not willing to use GCC macros, they can simply be disabled and since they only add sugar with no extra functionality, LURC retains the exact same set of functionality with or without this sugar, which we feel is important.

A.4 Related work

Several C libraries offer multi-threading facilities. LURC is a mixture of several ideas found in older libraries with its differences. Here are the most notable ones:

A.4.1 POSIX threads

POSIX threads are the most common type of threads on UNIX systems. They offer asynchronous threads that run in parallel, mutex objects for critical sections, condition variables for thread synchronisation, and good performance on most systems. Threads can only run in an asynchronous way, thus always needing both a broad knowledge of all the threads involved in the program and explicit protection of critical sections.
LURC offers synchronous threads as the default type of thread, but allows its threads to become asynchronous (and back) if they need to, using POSIX threads for that purpose, and thus giving the LURC thread to the POSIX thread scheduler during this period. We believe the two layers are complementary and some behaviour simply cannot be done in a cooperative way, even though in most cases it can.

A.4.2 Event loops

While this is not a typical threading model, event loop programming is the method of choice for programming applications in which several parts of a program are allowed to run concurrently, via an event scheduler. The analogy between an event loop and reactive programming is a good one: the reactive instant is an event loop iteration, during which reactive signals (or event loop events) are emitted to allow reactive threads (or event loop callbacks) to execute.
The main difference however is that event loops are limited to callbacks without state (the callback must terminate and exit every time it is called) whereas reactive threads can cooperate in the middle of a computation, and thus keep a state between activations. More than that, LURC integrates all event loop facilities via different types of signals and thus event loop programming interacts alongside reactive programming in LURC.

A.4.3 GNU/Pth

GNU/Pth is a cooperative threads library written in C, aimed at being efficient and portable on all UNIX systems. Like LURC, it features an Event Loop integration, and more types of signals than LURC. However it has only one type of cooperative thread (those using a different stack), and does not support asynchronisation. It also does not support control blocks like suspension, preemption and protection.

A.4.4 Fair Threads

The FairThreads [Bou04b] library in C is something that LURC is very close from: it features suspension, preemption and protection blocks, signals and even thread asynchronisation. Fair threads are available in several languages, all with variations of syntax, features and implementation, but with the same model in mind, and one that is very close to LURC. Fair threads support two types of threads: threads with state, which are always mapped to a native thread (most likely asynchronous Pthread) kept synchronous by the fair thread scheduler and library. This allows them to become asynchronous as we do in LURC, but has the disadvantage to be the default type of thread, even for threads we know will not be made asynchronous, which imposes (for example) a limit on the number of threads. LURC on the other hand supports two types of cooperative-only threads which do not require any native thread.
The other type of thread is an automaton, which does not map onto a native thread, and makes scheduling of several automata fairly efficient, but does not allow any state to be on the heap when cooperating (for example, it is not possible to cooperate within a function call). LURC on the other hand does not support automata threads: the main advantage of not needing a stack to execute is also its main drawback since it is impossible to cooperate within a function call. While we could have another type of LURC thread representing automata, a difference in semantics between threads would arise, and it is not the path we have chosen.
The most up-to-date version of FairThreads in C is called LOFT: a Language Over FairThreads [Bou03]. It is a syntactic extension to C with reactive primitives.

A.4.5 Reactive ML

RML [MP05] is a syntactic extension to the OCAML language, similar to the FairThreads and LOFT, but without FairThreads automata and with a type system integrated with the reactive primitives.
In RML like in LOFT, the reactive layer and the ML layer are distinct, even though it seems reactive code can be mixed with ML code in a reactive function (that is, a function explicitly declared as reactive). No reactive code is allowed in pure ML functions. However regular ML code in a reactive function is actually transformed into RML runtime function calls (the `if` statement becomes a `rml_if` function call), while the code is separated into different functions at cooperation points. This avoids having to save a continuation like in LOFT, but without real automata and an added cost at runtime.

A.4.6 ULM

Mainly inspired from FairThreads, ULM is a set of primitives for a reactive and mobile language. ULM specifies the deterministic scheduling rules that apply to ULM threads,

		NPTL	GNU/Pth	LURC			
				SC	SJ	AL	AJ
Time	Scheduling A	3.6s	1.5s	1.6s	1.6s	2s	2s
	Scheduling B	2s	1.5s	1.6s	1.6s	2s	2s
	Scheduling C	27s	1:25m	12.1s	11.5s	29.3s	11.7s
Memory	Virtual	8m	14m	1.8m	8m	8.2m	8.2m
	Residual	2m	2m	600k	1.9m	2.2m	2.2m

Table A.1: NPTL, GNU/Pth and LURC

and these primitives can be integrated in any language. LURC is based on that model, minus mobility. We believe the Reactive primitives to be worth basing a threading library on, as they allow a lot of derived functionality. Doing mobility in C is a whole different problem than threading, and thus was not included in the goals of the LURC project.

A.5 Benchmarks

In order to verify our results regarding two of our main objectives: being light and fast, we present several tests comparing LURC several other threading libraries. All tests have been made on dual core intel hardware with 1gb of RAM, under linux 2.6.20 from the feisty Ubuntu distribution.

A.5.1 Pthread and GNU/Pth

The first test compares the different types of LURC threads with the current Linux implementation of Pthread (NPTL) and the latest version of GNU/Pth (2.0.7). While GNU/Pth and LURC have both a model of cooperative threads, NPTL has asynchronous threads. What we wanted to test first is: can they be compared in terms of speed and memory usage? In order to stress the system on the number of threads and memory usage, we create 200 threads with a stack size of 32k and make them work for 50,000 iterations each. The results are presented in Table A.1 for NPTL, Gnu/Pth and LURC's Synchronous Copy, Synchronous Jump, Asynchronousable Lock and Asynchronousable Jump thread types.
The time results are split in three scheduling strategies. The Scheduling A was done with no constraint about scheduling: we just want to have the 200 threads do the 50,000 iterations in any order. For the Scheduling B, we want each thread to do his 50,000 iterations in one mutually exclusive step, before the next thread could do his iterations. For example, given the threads A to Z doing iterations 1 to 10 the scheduling should be: A(1)...A(10) ... Z(1)...Z(10). And in Scheduling C the scheduling is further constrained so that each thread iteration should be interleaved. Following on our example, the scheduling would be: A(1)...Z(1) ... A(10)...Z(10).
For Scheduling A we notice that the default scheduling of GNU/Pth and LURC (which we observe is the same) gets the job done faster than the preemptive scheduling of NPTL. This proves that having cooperation between threads done at the right place can be faster as trying to run everything at once. This is even more relevant when you know that the test machine is a dual-core processor, so both GNU/Pth and LURC are using half the processing power that NPTL is using. We also notice that GNU/Pth and LURC have

comparable performance, albeit lower for LURC's Asynchronous threads (even though they are not asynchronous in this test).
For Scheduling B we confirm that a well chosen cooperation point makes a difference for NPTL, which attains comparable performance to both other threading systems. As we are not aware of the semantics of GNU/Pth scheduling we are using the same system of mutex and condition variable to enforce the scheduling we want for both NPTL and GNU/Pth. In practise we observe that GNU/Pth's default scheduling is the same LURC for this test, and removing the mutex and condition variable has no time gain.
For Scheduling C it is clear that LURC is faster than both other systems. In fact what is interesting is that for the Synchronous Copy the time spent copying the stack at each cooperation is negligible. We also notice that both types of threads that jump to the next thread (as opposed to requiring an OS context switch) are almost three times as fast as the NPTL and LURC's Asynchronous Lock thread types. The difference can be explained by the cumulative costs of mutexes, condition variables and OS context switching. Since cooperating in an Asynchronous Lock thread involves the same mechanisms as in NPTL the performance is roughly the same. The time shown for the GNU/Pth test is actually a run without a mutex and condition variable, because using the same mechanism as the NPTL test here takes about 55 minutes. Instead of this mechanism, we noticed that the actual behaviour of cooperating with `pth_yield` in place of using a mutex and condition variable produces the same scheduling as LURC (whose scheduling behaviour is enforced by semantics), so we used this mechanism in the (realistic) hope that semantics can be deducted from GNU/Pth for thread cooperation scheduling.
As for the virtual and residual memory usages, since all threads have the same stack size of 32k they are all comparable except for LURC's Synchronous Copy type which saves in memory only the portion of the stack which is effectively used. And in such a test the effective use of the stack is low, thus explaining the low space required to save the threads which are not currently running.
We think these tests show that choosing a scheduling (when it is possible) can make a difference. The random preemptive scheduling of NPTL is not better than a deterministic scheduling for all applications, even with twice the CPU power. We also see that the Asynchronous Jump scheduling when cooperative pays off in terms of time, and the Synchronous Copy model of stack saving pays off in terms of memory.

A.5.2 LOFT and RML

We have also tested the closest reactive threading library: LOFT and RML. Although the semantics of these three threading systems are very close, some differences will explain the results of these benchmarks. The most important difference between LURC and RML/LOFT is that LURC is merely a library used in the C language, while LOFT and RML are language extensions over respectively C and ML. Both LOFT and RML separate strictly the reactive layer and the C/ML layer, which means that C or ML functions cannot have reactive instructions such as cooperation. As we have already described in Section A.4, both these languages compile their threads in a form of automaton that does not require stack space, thus greatly reducing the cost in terms of memory.
In our first set of benchmarks, we have tested four use cases for cooperative threads: creating a large number of threads (100000) interacting for a short time, making a small number of threads (1000 first, then 300) cooperate with one another for a long time, and at last the reversed waterfall. The reverse waterfall is a classic reactive test in which many threads

		LURC minimal			LURC maximal			LOFT	RML
		SC	SJ	AL	SC	SJ	AL		
Number of threads	Mem	54360	NA	NA	68460	NA	NA	33184	**30672**
	Time	**23.694**	NA	NA	27.58	NA	NA	121.99	54.624
Cooperation 1000	Mem	992	7464	NA	1172	7500	NA	**668**	1468
	Time	21.772	33.644	NA	25.834	39.036	NA	**6.93**	49.49
Cooperation 300	Mem	**552**	2500	2944	700	2604	2996	**??**	1216
	Time	6.382	7.064	27.398	7.102	10.326	27.896	**2.44**	13.428
Reverse Waterfall	Mem	1576	7684	NA	2016	8068	NA	**1036**	1540
	Time	**3.388**	4.108	NA	**3.37**	4.588	NA	59.752	9.352

Table A.2: Miscellaneous benchmarks

are created, all linked together so that the last created thread will emit a signal which will be propagated by all threads back to the first one.

In addition to RML and LOFT we have tested LURC in minimal and maximal configurations: in the minimal configuration LURC is compiled with a strictly minimal set of features in order to perform the benchmark, while in the maximal configuration all features are enabled. For each configuration we tested the three LURC thread types: Synchronous Copy, Synchronous Jump and Asynchronous Lock (in synchronous mode of course). The results are presented in Table A.2 with the emphasised numbers showing the best scores in memory usage in kilobytes (as residual memory, i.e. maximum memory really used as opposed to allocated but unused memory) and execution time in seconds. For both measurements lower is better.

The Not Applicable columns for LURC mean that the test did not succeed due to excessive memory requirements of the underlying types of threads: for both SJ and AL threads we have to allocate a stack with a fixed size, and creating so many threads requires too much memory. It is worth noting that for the Cooperation 300 test the resident memory for SJ and AL threads is around 3mb while the allocated memory hits the limit of 32bit addressing at 2.4gb, which is why we were unable to create more than 300 AL threads, and only a bit more SJ threads. The ?? mark for the LOFT Cooperation 300 memory test is due to a measurement problem which prevented us from measuring the memory but based on the Cooperation 1000 we can speculate that it would be the lowest.

Concerning LURC threads, we notice that there is only a small difference between the minimal and maximal configurations, so while the scheduling code is much more complex with every feature enabled, the price is not so great when these features are not used. Another thing to note is that when threads have a very small stack (this is the case in all four tests), cooperation time is not faster for SJ threads than SC threads, so the SC type of thread really makes sense for small thread stacks.

Comparing now LURC to LOFT and RML, we notice that in minimal memory usage (the SC threads are comparable to LOFT automata and RML continuations in that they don't allocate a fixed space for the stacks) LURC uses at most twice the memory of the best score. This is not so bad, but nothing to be proud of. Concerning speed now LURC is considerably faster for the first and fourth tests, and second to LOFT on the cooperation test. This is where automata really pay off and LOFT annihilates other players in this test.

	LURC	LOFT	RML
Avg	**0.29**	0.64	0.43
0-20%	**0.08**	0.15	0.13
20-40%	**0.23**	0.46	0.42
40-60%	**0.40**	0.81	0.72
60-80%	**0.48**	1.11	1.00
80-100%	**0.61**	1.46	1.40

Table A.3: Benchmarks Fredkin 500*500

A.5.3 Cellular Automata

For the second set of tests, we chose another well known benchmark for reactive threads: the Fredkin cellular automata [Bou04c]. This automata has cells become alive when their 8 closest neighbours are in odd number, or die otherwise. This is done in a 500 by 500 grid, by having each cell as a thread, which waits for its neighbours to notify it of their status. Each reactive instant gives us a generation. The results of this test are presented in Table A.3 and presented as follows: for each threading system we present the average duration of an instant, then the average instant duration depending on cell population (both in seconds). This is meaningful because the more cells are active (not waiting) the more cooperation is going to happen and the longest the instant will take to complete.
These results clearly show how LURC is faster for this type of application. The code base for the three tests are strictly similar, but for the propagation of valued signals. LOFT and RML both offer valued signals as part of their language, and a primitive that collects them by notifying each thread every time a single value is emitted, which can be several times in an instant. LURC does not support valued signals, and follows the path of ULM in which valued signals can be built in an ad-hoc manner on top of plain signals, without special support from the scheduler. The idea is that if a scheduler supports valued signals it can only support one flavour of them, whereas there are many different uses for valued signals, all of which can be built on top of plain signals.
This is probably what made the difference here: we were able to construct a valued signal that does not aggregate values but just counts its number of emissions during the instant, and delivers that value only once, at the next instant. Having an emission count is enough for the Fredkin automata, where we just want to count the neighbours, and being awakened only once per instant in order to count them is crucial to reducing the number of cooperations.

A.6 Future directions

We present several areas where some work is needed in LURC, ranging from increasing efficiency of the Reactive Event Loop with a different implementation, to designing new semantics that would enable LURC to benefit from multiple processors or cores.

A.6.1 Reactive Event Loop

The REL in LURC is implemented by associating file descriptors or timeouts to LURC signals. These signals are then collected together and used in a `select` call during the End

Of Instant phase. On several systems a better, faster, more scalable approach exists: `epoll` on Linux, *kqueue* on FreeBSD or */dev/poll* on Solaris. We think that making use of these mechanisms instead of `select` is straightforward for LURC, and it would actually allow LURC to benefit from all their features. For example, it would be easy to register in the event set when creating the LURC REL signal instead of rebuilding the event set for each `select` equivalent.
There are other external asynchronous events which would be interesting for LURC to integrate, such as POSIX signals. POSIX signals are usually integrated in an event loop by associating them an internal pipe and having a signal handler write on that pipe, so that calls to `select` would be notified of the signal emission. This is one way LURC could implement POSIX signal integration. An simpler way seems to be in the works at least on Linux, in the form of `signalfd`: a new system call which maps a POSIX signal emission to an event on a file descriptor. This would be perfect for integration in the REL.
The work on integrating asynchronous events and event loops on Linux may also improve LURC's handling of timeout signals. A new system call `timerfd` maps a timer event to a file descriptor. The way LURC handles timeout signals is by calculating the closest timeout on each call to `select` and again calculating all those which have been triggered after the select call. Using `timerfd` and `epoll` it would be much easier and more efficient since all the timeouts would be handled by the kernel.

A.6.2 New semantics for SMP

While working on adding LURC as a backend for STklos' [Gal] threading library, we noticed several drawbacks that should be addressed in future releases of LURC.
The first problem we encountered was that once you expose the possibility of detaching a thread and becoming asynchronous, we are faced with a dilemma: in a way it makes sense that the synchronous reactive API does not apply anymore, but that also means that each library that has deal with scheduling now has to deal with two possibilities: was this function called while synchronous, in which case it behaves this way, or was is called while asynchronous, and it has to behave differently. For example, you cannot write a function that waits for a LURC signal and expect it to work both synchronously and asynchronously. What is possible right now is to check the status of the current thread and determine whether it is synchronous or not, and behave accordingly, but that is a tedious task. We asked ourselves the question of why the reactive API of LURC doesn't make sense in an asynchronous thread? There has to be a way to give it a meaning. For example mutex locks: when a thread is asynchronous and has to work on data shared with other threads it usually has to use mutex locks. But this also makes sense for synchronous reactive threads: mutexes have to be used in case a call to a third-party function call triggers a cooperation in the middle of a critical section. It is trivial in LURC to implement mutex locks using LURC signals, but since LURC signals cannot be used while asynchronous, this is not good enough.
We think we can give a meaning to all the reactive API of LURC even while asynchronous. Our current view is that threads that detach themselves to become asynchronous should automatically be assigned their own synchronous scheduler. Becoming asynchronous would then mean creating a new synchronous zone, which runs asynchronously to other synchronous zones. Much in the way that ULM offers thread migration between different computers, we can treat these synchronous zones as synchronous places between which threads could migrate. This does not isolate the data, because memory is still shared be-

tween the zones, but if we also implement ULM's remote references we can then provide a safe way to exchange data between asynchronous zones. ULM references are data cells which are local to a location (in our case: a synchronous zone) or a thread, and whose reading or writing blocks the caller thread until the caller thread migrates to the location, or the thread to which it belongs comes on the same zone as the blocked thread.
So this means that when a thread becomes asynchronous, it automatically creates a new synchronous zone between which synchronous threads can migrate. The semantics for signal could be adapted in order to broadcast signals asynchronously between zones. For example signals would still be emitted intra-instantaneously in the current zone, and sent for emission in all other zones at the beginning of one of their local instant. In this way all the reactive API of lurc makes sense both synchronously and asynchronously.
The other advantage of this idea is that this solves our second problem: we can now take advantage of multiple processors easily while maintaining several totally deterministic synchronous zones (say one on each processor or core), while also allowing communication and thread migration between the zones.

A.7 Conclusion

We have presented LURC's features, its implementation and examples illustrating its possibilities and ease of use. We think having several implementations for threads that coexist under a deterministic semantics is a solution to balance the tradeoff between speed and semantics, while leaving the choice for some threads to separate from the cooperative scheduler when they need to provides a good flexibility without sacrificing the determinism of the other cooperative threads. We believe the set of features makes it easy to port applications from other existing thread models, while remaining highly portable on most POSIX/GCC platforms. We hope to have demonstrated that LURC applies not only to general purpose hardware, but also the world of constrained devices such as embedded platforms, where thread tuning can make a big difference in the adoption of a thread model. We also think the syntactic extensions help a great deal by making threading features appear to be a language extension which should have its place in C.
LURC is licensed under the GNU Public License, source and documentation are available on its website at `http://www.inria.fr/mimosa/Stephane.Epardaud/lurc/`.

Résumé

Afin de résoudre les problèmes liés à l'intégration d'un nombre croissant d'appareils programmables, nous proposons un langage d'agents mobiles. Ces agents mobiles sont capables de migrer d'un appareil ou ordinateur à l'autre afin d'exploiter au mieux ses ressources, ce qui permet de profiter au mieux des capacités de chaque appareil à partir d'un unique programme.
Ce langage est ULM: Un Langage pour la Mobilité. Nous présentons dans cette thèse ses fonctionnalités, ses particularités, ainsi que sa mise en œuvre. ULM est un dérivé du langage Scheme, auquel nous avons ajouté les fonctionnalités liées à la mobilité ainsi qu'à l'interaction entre les agents mobiles. ULM possède un ensemble de primitives permettant la création d'agents à mobilité forte, avec un ordonnancement coopératif déterministe, et des primitives de contrôles telles que la suspension ou la préemption faible.
Nous présentons dans cette thèse l'intégration de ces primitives dans le langage Scheme, ainsi que leur interaction et l'ajout de certaines nouvelles primitives telles que la préemption forte ou la migration sûre. Nous présentons ensuite la sémantique dénotationnelle du langage et sa mise en œuvre au moyen d'un compilateur vers code-octet, et de deux machines virtuelles: une écrite en Bigloo Scheme pour exécution sur des ordinateurs traditionnels, l'autre écrite en Java ME pour les téléphones portables. Nous présentons ensuite l'utilisation possible d'ULM comme remplacement de programmes écrits pour des boucles d'évènements, l'interface entre ULM et des langages externes, quelques exemples d'utilisation d'ULM, puis les travaux futurs avant de conclure.
Mots clefs: mobilité forte, agents, ordonnancement, langage, machine virtuelle, Scheme.

Abstract

In order to avoid the problems raised by the integration of a growing number of programmable home appliances, we propose a language with mobile agents. These mobile agents are capable of migrating from one appliance or computer to another in order to work on its local resources, which allows us to benefit from each appliance's capabilities from a single program.
This language is called ULM: Un Langage pour la Mobilité. We present in this dissertation its features, its differences with other languages, as well as its implementation. ULM is based on the Scheme language, to which we have added functionality linked with mobility and the communication of mobile agents. ULM has a number of primitives allowing the creation of strongly mobile agents, with a cooperative deterministic scheduling, and control primitives such as suspension or weak preemption.
We present in this dissertation the integration of these primitives in the Scheme language, as well as their interaction and the addition of new primitives such as strong preemption and safe migration. We then present the denotational semantics, and its implementation with a bytecode compiler and two virtual machines: one written in Bigloo Scheme for execution on traditional computers, the other in Java ME for mobile phones. We present then the possible use of ULM as a replacement for programs written for event loops, the interfacing of ULM and external languages, a few examples of ULM applications, and future work before we conclude.
Keywords: strong mobility, agents, scheduling, language, virtual machine, Scheme.