HAL open science

The FunLoft Language

Frédéric Boussinot

To cite this version:

Frédéric Boussinot. The FunLoft Language. 2010. inria-00497705

HAL Id: inria-00497705

https://inria.hal.science/inria-00497705v1

Preprint submitted on 5 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The FunLoft Language
(draft 1.0 / FL Compiler v0.2)

ANR-08-EMER-010 PARTOUT

Frédéric Boussinot
INRIA Méditerranée -INDES Team
2004 route des Lucioles - BP 93
F-06902 Sophia-Antipolis
frederic.boussinot@sophia.inria.fr

July 5, 2010

Chapter 1

Introduction

A Historical Perspective

Per Brinch Hansen presents the invention of concurrent programming as one of the major revolutions in computer programming [29]. This revolution initiated from operating system programming, in the mid 1960, and as P. Brinch Hansen says: *"The development of concurrent programming was originally motivated by the desire to develop reliable operating systems"*.

Maybe the most important problems in concurrent programming result from uncontrolled concurrent accesses to shared data. These accesses lead to so-called *time-dependent errors* (*data-races* which may possibly cause data corruption are just an example of them). Several constructs have been proposed at language level to provide means to avoid these errors, the most famous of them being the *monitor* concept of Brinch Hansen and Hoare. In [42] Hoare introduces monitors as follows: *"A primary task [...] is to construct resource allocation (or scheduling) algorithms for resources of various kind [...]. In order to simplify his task [the designer] should try to construct separate schedulers for each class of resources. Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. Such a collection of associated data and procedures is known as a monitor..."*.

Monitor-based constructs have been introduced in many high-level programming languages, among which are Concurrent Pascal and Modula. It is interesting to notice that Java [14] also belongs to this family of languages (see however [28] for a without-concession analysis of Java parallelism). By controlling the ways monitors are used, a compiler should be able to statically check access rights of concurrent programs, an idea proposed in 1971 by C.A.R. Hoare [41]. However, in such an approach, concurrent programs should only communicate and synchronise using monitors: direct communication through pointers or references should be avoided. Note that this is not the case in Java, where thread communication through references is allowed; as a consequence, Java compilers

are unable to statically detect time-dependent errors.

Various alternatives to monitors have been proposed for communication and synchronisation, among which are message passing and rendez-vous. In message passing, the sender sends a message to a receiver without being blocked until reception, and received messages are stored in mail-boxes. This model of communication originates from the actor model of Carl Hewitt and is at the basis of the Erlang programming language [13]. As the sender of a message has no means to be notified when the message is received, synchronisation between concurrent programs must be based on user-defined protocols, which makes it quite complex. In rendez-vous-based communication, two concurrent programs synchronise to perform a common communication action. The Ada programming language is based on this model. Rendez-vous is also used in the language Concurrent ML [57] which extends ML with concurrent primitives.

Threads and Processes

The development of operating systems has strongly influenced the design and the implementation of programming languages. The most famous example of related system and language is the couple Unix/C. Amongst the notions that were first introduced in operating systems and became later reflected in programming constructs are the ones of process and thread. Processes appear as concurrent system units with run-time memory protection. Processes have been introduced in many concurrent languages as denoting concurrent programs with private memory and structured communication means (for example, in Erlang, Occam, or CML).

As opposed to processes, threads, which are sometime called *light-weight processes*, do not have built-in memory protection and have direct access to a shared memory, a single address space in the terminology used by Birrell in [17]. Here is the way he introduces threads: *"Having the threads execute within a "single address space" means that the computer's addressing hardware is configured so as to permit the threads to read and write the same memory locations. In a high-level language, this usually corresponds to the fact that the off-stack (global) variables are shared among all the threads of the program. Each thread executes on a separate call stack with its own separate local variables. The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer."* In other words, threads raise severe time-dependent problems, the solution of which is generally left to the programmer. From that point of view, threads are much more difficult to use than processes. At user level, threads are available under the form of a library in many languages (C, C++, CAML, for example) implementing the Posix [53] norm and are sometimes directly incorporated in languages like Java.

A crucial point is that threads are more efficient than processes. In [17], Birrell indicates: *"With (multi-processor machines), there really are multiple simultaneous points of execution, and threads are an attractive tool for allowing a program to take advantage of the available hardware. The alternative, with most*

conventional operating systems, is to configure your program as multiple separate processes, running in separate address spaces. This tends to be expensive to set up, and the costs of communicating between address spaces are often high, even in the presence of shared segments. By using a lightweight multi-threading facility, the programmer can utilize the processors cheaply". The advantage of threads over processes also comes from the possibility to implement threads directly at user-level and not in the operating system kernel, as is the case for processes. Moreover, threads can communicate directly (with care, however) through the shared memory, with minimal extra run-time overhead.

Scheduling Strategies

The possibility of interrupts and the presence of processes in operating systems lead to "non-deterministic" behaviours. Non-determinism basically comes from the fact that process statements are interleaved by the processor. For example, a process which executes an input/output operation becomes suspended, until an interrupt is produced to indicate that the operation is completed; thus, statements belonging to other processes can be run during suspension. As another example, a process whose execution takes too much time should be preempted by the operating system and control should be given to other processes; preemption is also a source of non-determinism as the global behavior depends on the system preemption strategy.

At an abstract level, input, output, and concurrency have been put together by Hoare in a model of computation named *Communicating Sequential Processes* [43] (CSP). A parallel operator is introduced which specifies concurrent execution of sequential commands called processes. *"All the processes start simultaneously, and the parallel command ends only when they are all finished. They may not communicate with each other by updating global variables"*. In CSP, non-determinism appears in the form of a variant of Dijkstra's guarded commands.

Threads are controlled by the system by means of a scheduler. The scheduling strategy is preemptive if the scheduler can suspend a thread at any time. The strategy is cooperative if the points where suspension can occur are explicitly delimited. Cooperative threads are often rejected because a lack of cooperation (for example, a loop without any system call in it) can block the whole system, the looping thread taking all the computing resources and preventing the other threads form running. However, cooperative threads are more efficient than preemptive ones: context-switches are under the control of the user, which can thus limit their number; moreover, context-switches can be implemented directly without any kernel functionalities.

Security Issues

In [29] P. Brinch Hansen expresses the fundamental security property for concurrency: *"For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do*

not interfere with each other in time-dependent ways". This non-interference property is at the basis of the notion of process as it appears in modern operating systems. In programming languages, non-interference should be checked by compilers. This task is achieved rather simply when inter-process communication can be syntaxically controlled. Concurrent Pascal [27] is the first language to do so, by defining scope-rules for variables allowing the compiler to reject any sharing on these variables. Inter-process communication is achieved in Concurrent Pascal with monitors, that are also controlled by the compiler for non-interference. Low-level access to the memory is not possible in Concurrent Pascal, which means that there can be no direct inter-process communication through the shared memory (using references, for example).

When direct access to shared memory is possible, it becomes a non-trivial task to control non-interference. Java is of course insecure from this point of view, as threads can interfere through shared references.

The implementation of CML cannot be considered as secure either, since, as Reppy says: *"CML is a message-passing language, which means that processes are viewed as executing in independent address spaces with their only communication being via messages. But, since SML provides updatable references, this is a fiction that must be maintained by programming style and convention"* [57] (p. 39).

The Cyclone language [61] is a safe variant of C (safety here means that no run-time crash can occur); safety is achieved using a combination of static analyses and inserted run-time checks. The extension to multithreading is considered in [37] where data races are tackled, but no real programming language has been yet designed on this basis.

Other security measures are specific to parallel programming. For example, an important security measure is to check that processes are not created in an uncontrolled way, during the system life-time. A system that would keep creating new processes without eliminating old ones would eventually run out of memory and crash. These checks basically have to do with the control of loops and recursive calls.

In the context of message passing, it seems important that messages do not continue to accumulate in receivers' mail-boxes. As mail-boxes are data structures, this kind of checks have basically to do with controlling that data are of bounded size. In Erlang, this task is left to the user: *"...as any messages not matched by receive are left in the mailbox, it is the programmer's responsability to make sure that the system does not fill up with such messages"*. [13] (p. 69, 2nd Edition). Synchronous languages considered in the next section propose a solution to this problem.

Synchronous Languages

In 1971, Dijkstra made a rather surprising statement: *"One of the primary function of an operating system is to rebuild a machine that must be regarded as non-deterministic (on account of cycle stealing and interrupts) into a more or less deterministic automaton"*. [32]. It is surprising because it relates parallelism

and determinism. Twenty years after, were designed several languages mixing parallelism and determinism in a semantically sound way. These proposals belong to the family of synchronous languages [38].

In synchronous languages, concurrent programs are processes that share a logical clock, the units of which are called *instants*. All processes automatically synchronise at each instant.

Lustre [30] adopts a data-flow programming style; the underlying model is the one of G. Kahn [45] in which sequential processes execute in parallel and communicate through unbounded "first-in/first-out" channels on which reading is blocking. This is the first model of parallel computations which has received a formal semantics (in the denotational style). This model is inherently deterministic: histories (i.e. possibly infinite sequences of values) of output channels depend only on histories of input channels (and in particular not on their timing). Lustre introduces syntactic constraints, and a mode of execution adapted to instants, to execute programs in bounded memory: a channel always contains at most one value in it. Amongst the constraints is the static nature of networks: processes or channels cannot be dynamically created. In the variant Lucid Synchrone [55] of Lustre, functions are first-class values and can thus be communicated through channels (what is forbidden in Lustre).

In constrast with Lustre, the programming style of Esterel [16] is imperative. The parallel operator is explicit and is an adaptation to instants of the one of CSP: the parallel branches are executed simultaneously, at each instant, and the parallel operator terminates at the instant where all its branches terminate. Communication of parallel processes uses broadcast signals to which data can be associated. At each instant, a signal is consistently seen as present or absent by all the parallel processes, which also all read the same associated data. In this way, time-independency and determinism are assured for signals. Direct communication of parallel processes through shared variables is forbidden, and this interdiction is statically checked. However, as procedures and functions are not defined in Esterel but in a "host" language (basically, C), there is no way for Esterel compilers to verify that no communication could appear in this way; that is, compilers are unable to verify the absence of time-dependent errors.

An important characteristics of synchronous languages is their formal semantics: the operational meaning of programs can be derived from the meanings of their constituents. The semantical framework is that of Plotkin's *Structural Operational Semantics* [54], based on rewriting rules. According to the semantics of Esterel, for example, the meaning of a parallel program is a deterministic automaton describing its input/output behavior through signals; this automaton, in which parallelism has been compiled away, is actually the exact output of one of the first Esterel compilers (the v3 version, described in [23]). The statement of Dijkstra cited at the beginning of the section seems now not so surprising; even more: the automaton is not "more or less" deterministic, but it is totally deterministic!

Reactive Programming

Strongly related to synchronous languages is a family of formalisms grouped under the name *Reactive Programming* [8]. The language Reactive-C [18] is historically the first member of this family. The purpose was to allow programming of reactive systems in C, in a way which is close to Esterel, but without its limitations, and especially its static character: in Reactive-C new instances of reactive procedures can be dynamically created at any time. Technically, this becomes possible because one has delayed to the next instant the reaction to the absence of a signal, in order to avoid the so-called "causality cycles" of Esterel. Typically, a causality cycle appears when a signal is emitted during one instant in response to its detection as absent during this very instant. The broadcast property would be violated and time-dependent behaviors would appear if causality cycles were accepted. A variant of Esterel, called SL [24] based on the rejection of immediate reaction to signal absence has been designed and implemented in Reactive-C.

Reactive programming is also implemented in SML [56], in Java (SugarCubes [26], FairThreads), in ML (ReactiveML [51]), in C (Reactive-C, FairThreads [21]), and in Scheme [60]. Most of these formalisms are presented in [8].

The formalism of FairThreads, built over C, deserves a special status amongst the reactive programming formalisms for two reasons:

- In FairThreads, concurrent programs are basically threads (called fair threads), while the other formalisms use the standard parallel operator, adapted for instants. The justification of the use of threads is efficiency: first, fair threads, just like other threads, can directly communicate through the shared memory without overhead; second, fair threads can benefit from multi-processors or multi-core architectures (which is problematic with processes; see Birrell's previous remark).

- Several logical clocks can be used simultaneously in the same application, rather than a unique one as with the other formalisms. Each clock is associated with a scheduler which thus controls the fair threads linked to it. The various schedulers are run autonomously and asynchronously. Fair threads have the possibility to dynamically migrate from one scheduler to another scheduler. They even have the possibility to become autonomous, being then only under the control of the operating system, just as standard threads. Such autonomous fair threads are called *unlinked*.

FairThreads is of course not secure, in the previous sense; all kinds of time-dependent errors can occur and the compiler does not even try to capture some of them. The reason is that this would mean tracking accesses through shared pointers by unlinked threads. As FairThreads is built on top of C, this is certainly an extremely difficult task (have a look to [37] to get an idea of what this means in the context of Cyclone). Clearly, FairThreads, like C, is not the good formalism when security is searched for.

The previous statement can be however relativised when there is only one scheduler to which all threads are linked (no unlinked threads). Indeed, in this case, all fair threads are scheduled cooperatively by the scheduler and this mode of execution prevents data-races and time-dependent errors. Operations executed between two cooperation points are executed atomically; there is no risk that an other thread could interleave its execution with them, by definition of the cooperative scheduling, and thus there is no risk of time-dependent errors.

***

The pioneers' work cited previously has raised the need of abstract notations for expressing concurrency and communication. The notion of a process with its associated private memory has emerged as a central notion. Monitors have been defined to provide processes with a basic interface with memory and have appeared to be appropriate for process communication. Monitors can be viewed as an elementary tool with which one can implement various abstract communication protocols. From the start, the emphasis has been put on static detection of interference in process accesses to memory. When monitors are the only possible communication means, checking for non-interference is achievable by compilers.

A multitude of proposals have been made, studied, and implemented for process communication, including asynchronous message-passing, synchronous rendez-vous, and broadcast signals. Signal broadcast is very natural in the context of concurrency; intuitively, it corresponds to radio communication with one emitter and several receivers always getting the same information. This form of communication is modular: adding in a process a new signal receiver, or a new signal producer is totally transparent to the other processes. In presence of instants, signals offer a good alternative to monitors for process communication and synchronisation.

Expressivity of these proposals has been tested on standard examples such as Dijkstra's dining philosophers [32], and various producer-consumer problems. Despite the fact that expressivity is not a well-defined notion, it appears that each of these proposals is as expressive as the others; it is quite a matter of taste to choose one of them. However, from an implementation point of view, the various proposals are not equivalent.

None of the synchronous languages nor of the reactive programming formalisms can be considered as secure as none of them verifies the fundamental security property stated by P. Brinch Hansen. Some of them allow for partial checks of non-interference (only variables are controlled in Esterel, for example), but most of them don't provide any checks.

From that point, two questions come naturally: first, how to make synchronous languages and reactive programming formalisms secure (in the sense of Brinch Hansen)? second, is it possible to reconcile shared memory communication and security?

The FunLoft Proposal

FunLoft is a proposal for answering the two previous questions. The underlying model is the one of FairThreads, thus based on the existence of a shared memory. But, accesses to the shared memory is controled in a way which forbids time-dependent errors: FunLoft is a safe language. Moreover, threads have the possibility to execute in a synchronous way and then to communicate through broadcast signals (events): FunLoft belongs to the synchronous-reactive family of languages. An important point is that the reactive engine based technique defined for implementing the formalisms of the reactive approach can also be used for FunLoft: FunLoft can be implemented very efficiently. Morever, FunLoft programs can exploit multicore architectures, which is another way to increase program efficiency.

Ideally, a model for concurrent programming should have the following properties:

1. Time-dependent errors should be statically detected.
2. There should be abstract means for concurrent programs to communicate and synchronise.
3. A precise and simple semantics should exist for concurrent programs.
4. Safety should be achieved: no run-time crash should be possible.
5. There should be means to benefit from multi-processor architectures.
6. Communication and synchronisation should be as efficient as possible.
7. Systems with a large number of concurrent programs should be implemented efficiently.

From previous section, synchrous languages and reactive programming formalisms are not fulfiling all these characteristics. Basically, time-dependent errors are not statically detected (item 1). Formalisms with no possible direct communication through shared memory but only abstract means for process communication (either based on monitors, mail-boxes, rendez-vous, or signals) raise a problem of efficiency; basically, they don't fullfill item 6. Formalisms with possible communication through shared memory (at langage definition level or at implementation level) are basically insecure. Moreover, it is very difficult to give them simple and precise semantics (item 3) due to the large-scale non-determinism resulting from concurrent accesses to the memory.

Structure of the Text

The basic model of FairThreads and the syntax of FunLoft are overviewed in Chapter 2.

The language is introduced in Chapter 3. Definitions of variables, types, functions, external definitions, schedulers, and modules are presented in Section 3.1. Values and expressions are considered in 3.2. Instructions are described in 3.3.

Static checks are described in Chapter 4. Reactivity, memory separation, resource control, and control on parametric types are considered in turn.

The programming style indiced by the language is considered in Chapter 5. The following issues are more precisely considered: nondeterminism, data protection, and division by zero and array out-of-bounds errors.

The language is used to code several standard examples regrouped in Chapter 6. An example of mutual stop is considered in 6.1. Several traditional examples of synchronisation, readers/writers, and producer/consumers are coded in Sections 6.2 to 6.5. A small reflex game (a traditional example of reactive system) is coded in 6.6. Finally, a graphical example of two synchronised threads illustrating the equation *Sine* + *Cosine* = *Circle* is given in 6.7.

Chapter 7 considers cellular automata and their coding in FunLoft. The well-known Game of Life is considered in 7.1, and self-repliacting loops (more precisely, the ones of Sayama) are considered in 7.2.

The simulation of colliding particles is considered in Chapter 8. This simulation example shows the benefit of using several (actually, 2) synchronised schedulers.

Chapter 9 describes a prey/predator simulation, reusing the particles defined in Chapter 8. The simulation shows an aspect of dynamic creation of threads (new fresh preys are dynamically introduced when all have been killed).

Dataflow programming is considered in Chapter 10. Static checks to control the memory usage have to be switched-off, in order to implement dataflow systems with FunLoft. Channels are described in Section 10.1 and processes in 10.2. Then several examples of systems producing sequences of numbers are coded: Fibonnacci numbers in 10.3, prime numbers in 10.4, lucky numbers in 10.5, and numbers that are both prime and lucky in 10.6. The last example is another illustration of of multicore programming in FunLoft.

The compiler of FunLoft is overviewed in Chapter 11. Related Work is described in Chapter 12. Finally, Chapter 13 concludes the text.

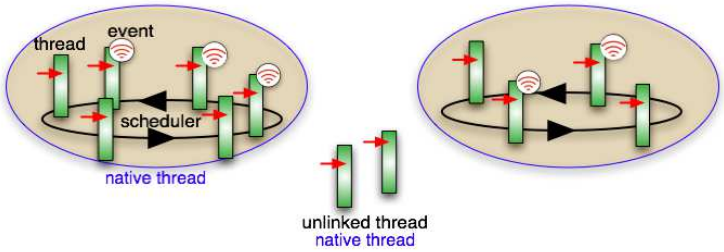
Chapter 2

Language Overview

To describe the basic computing model on which a programming language relies is often a good way to introduce it. In the present case, the model is basically the one of FairThreads[1]. One thus starts by an intuitive description of FairThreads before introducing the syntax of FunLoft.

2.1 Overview of the Model

The FairThreads model of FunLoft is basically made of threads and schedulers. Maybe the simplest way to describe the model is to comment on some drawings; the first is this one:



In this drawing, *schedulers* are symbolized by a colored oval, and *threads* are symbolised by a rectangle pointed by a red arrow (symbolizing the "program counter" of the thread). The drawing shows a system made of two schedulers to which several threads are *linked*. The threads linked to a scheduler are under its control: actually they are run *cooperatively*, that is they receive the control from the scheduler and, after having performed some computation, they return the control back to the scheduler which distributes it to another thread, and so on forever.

[1]Stricly speaking, the model presented here is an extension of the initial model of FairThreads presented in [21].

Instants

The execution of the threads linked to the same scheduler is divided in *rounds*: all the threads must have cooperated during the current round, and nothing should exist that could make the system progress, before the next round can start. This kind of execution is called *synchronous* and the rounds are called *instants*. One can see instants as defining some kind of logical clock; then, all the threads linked to a scheduler are executing at the same pace, according to the scheduler clock. Indeed, a linked thread that has finished its job for the current round will receive the control back from the scheduler only after all the other linked threads have also finished their work for the current round.

Linked threads have the possibility to synchronize and communicate using *events* (sometimes called *signals*) which are broadcast by the scheduler to all the linked threads. At each instant, an event is either absent or present, and this presence status is the same for all the (linked) threads. Moreover, values can be held by events, and all the threads see exactly the same values.

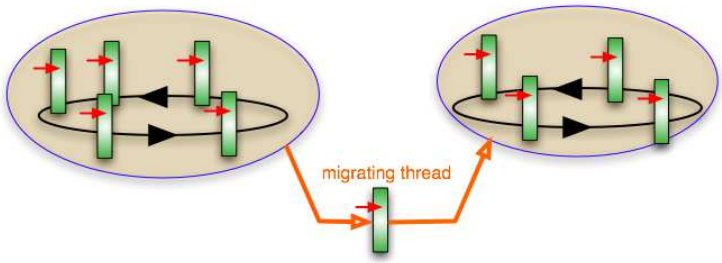
The model asks for a property of schedulers, called *reactivity*: no scheduler should stay stuck on one instant, being unable to proceed to the next instant. This implies that all linked threads are actually cooperating, as otherwise the absence of cooperation of one thread would prevent the termination of the current instant, and thus would prevent the scheduler to proceed to the next instant.

Unlinked Threads

The drawing also shows two *unlinked* threads which are not linked to any scheduler. These threads are run asynchronously and in a *preemptive* way by the operating system (OS): they can be preempted or given the control by the OS at any time. Note that cooperation has no meaning for unlinked threads (cooperation with what?). The main purpose of unlinked threads is actually to execute non-cooperative code (for example, a call to a blocking library function such as reading the keyboard). As unlinked threads, schedulers are executed in a preemptive way by the OS. Actually, schedulers and unlinked threads are implemented as *native threads* (sometimes also called *kernel threads*) which are to be clearly distinguished from previous threads (which are called *user threads*).

Migration

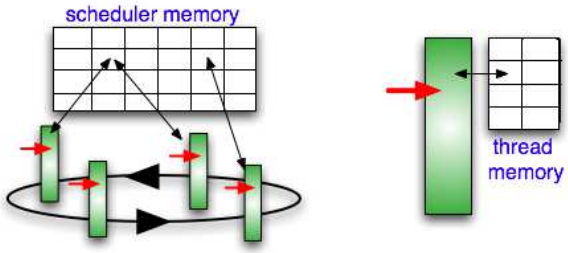
A thread linked to a scheduler can migrate to another scheduler to which it will link, as shown on:



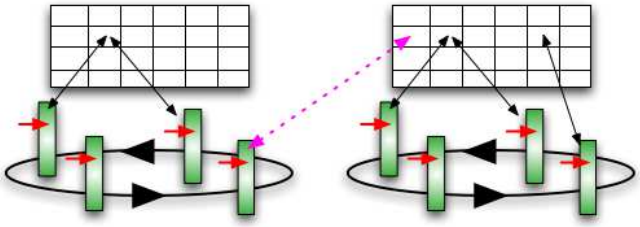
Threads are actually free to link and unlink at any time during their execution. The migration of threads is basically asynchronous as the instants of the two schedulers are not related. For example, a migrating thread can leave the source scheduler at instant n and reach the target scheduler at instant p, for totally different n and p.

Memory Structure

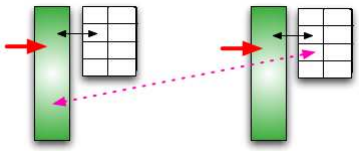
A memory is associated to each scheduler and to each thread, as shown on:



Two characteristics are asked by the model: first, the memory of a scheduler is only accessible (plain lines) by the threads linked to the scheduler. Thus, the access corresponding to the doted line in the following drawing, is forbidden:

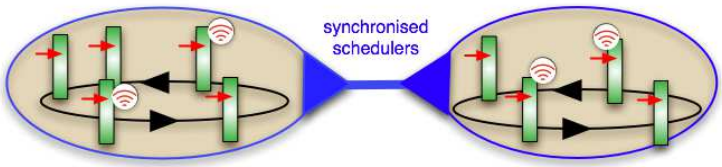


Second, the memory of a thread can only be accessed by the thread (independently of the status linked or unlinked of the thread). Thus, the access corresponding to the doted line in the following drawing, is forbidden:



Synchronised Schedulers

There is a possibility shown on the following drawing to *synchronise* schedulers.



Basically, to synchronise schedulers means to force them to synchronise at each end of instant. Thus, synchronised schedulers share instants and can be seen as plugged on the same logical clock. Note however that, during each instant, synchronised schedulers run asynchronously, that is autonomously. Synchronised schedulers share events, which is meaningful because they are sharing instants.

Absence of Interferences

The cooperative mode of execution of the threads linked to the same scheduler makes programming easier, as the *atomicity* of the code run between two cooperation points is automatically preserved. Thus, up to cooperation, the programmer does not have to worry about the possibilities of *interferences* with the other threads. For example, reading twice the same memory location while evaluating an expression automatically implies that the read values are the same. Thus, noting `!x` the reading of a location named `x`, the expression `!x+!x` is always equal to `2*!x`, *which is not true in a non-cooperative context* where, during evaluation of the first expression, there is the possibility for another thread to change the value of `x` in between the two readings.

The structuration of the memory has a major consequence: the absence of interferences, true by construction for the threads linked to the same scheduler, actually extends to all the threads, either unlinked or linked to different schedulers. Systems are thus totally free of interferences, *which makes locks useless.*

Resource Control

A last component of the model should now be introduced: namely, the control on the two resources that are the memory and the CPU. The model demands these two resources to be *bounded*. More precisely, there should exist two functions of

the program inputs, one that bounds the global memory used by the program, and the other that bounds the amount of computing resource needed for passing from one instant to the next. Concerning the memory, this means that assuming that the size (in a way to be defined) of inputs is always less than n, then the size of the global memory needed to run the program is always less than $f(n)$, for a specific function f. Concerning CPU, this means that, at all instants, first, the number of threads that are actually present in the system is bounded; second, the CPU used by each thread is also bounded. Note that the exact nature of the bounding functions is left unspecified; only their existence matters.

Actually, the FairThreads model can be declined under two forms: a *weak* one, in which resource control is not considered, and a *strong* one which includes it. In the following, except when explicitely stated, one always considers the strong model.

Now, let's leave the model and turn to the syntax.

2.2 Overview of the Language

Let us now consider the language FunLoft[2]. A program is a list of definition of variables, types, functions, and modules, in a syntax inspirated from CAML [2]. In order to be executed, a program must contain the definition of a module named `main`, which is the program entry-point. A variable is a name to which a value is associated. Values are first-order only and range over standard values (booleans and integers, for example), values of user-defined types, as well as names of threads, events and schedulers. Type definitions introduce possibly recursive polymorphic structured datatypes, built from union and concatenation of other types. Functions can be recursively defined; however, recursion is checked for termination, in a sense explained later, and thus function calls always terminate.

Schedulers and Threads

A scheduler controls the threads which are linked to it and provides them with a portion of shared memory. A special scheduler (the *implicit scheduler*) is automatically launched by each executable program and a thread created as an instance of the main module is linked to it. The basic task of a scheduler is to schedule the threads which are linked to it. The scheduling is cooperative: linked threads have to return the control to the scheduler, in order to let the other threads execute. Leaving the control can be either explicit, via the `cooperate`, `get_all_values`, and `for_all_values`, or implicit, by waiting for an event (`await` statement) which is not present. All linked threads are cyclically considered by the scheduler until all of them have reached a suspension point (`cooperate` or `await`). Then, and only then, a new cycle can start. Cycles are called *instants*. A scheduler thus defines an automatic synchronization mechanism which forces all the threads it controls to run at the same pace: all

[2]This overview is in a large extend taken from [22].

the threads must have finished their execution for the current instant before the next instant can start. Note that the same thread can receive the control several times during the same instant; this is for example the case when the thread waits for an event which is generated (`generate` statement) by another thread later in the same instant. In this case, the thread receives the control a first time and then suspends, waiting for the event. The control then goes to the other threads, and returns back to the first thread after the generation of the event. At creation, each thread is linked to one scheduler (by default, the implicit scheduler). Several schedulers can be defined and can simultaneously run in the same program. Schedulers thus define synchronous areas in which threads execute in cooperation. Basically, schedulers run autonomously, in a preemptive way, under the supervision of the OS. However, it is possible to define *synchronised schedulers* that share the same instants (and also the same events; see below).

During their execution, threads can unlink from the scheduler to which they are currently linked (`unlink` statement), and become free from any scheduler synchronization. Such free unlinked threads are, like schedulers, run by kernel threads under the supervision of the OS. Threads can also dynamically move to a scheduler (`link` statement).

Modules

Modules are templates from which threads, called *instances*, are created. A module can have parameters which define corresponding parameters of its instances. A module also defines variables local to each created thread. As opposed to functions, modules cannot be recursively defined. The body of a module is basically a sequence of instructions with usual control statements such as `loop`, `repeat` statements, and conditional statements. There are two types of instructions: atomic instructions and non-atomic ones. Atomic instructions are logically run in a single instant. Function calls belong to this kind of instruction. Execution of non-atomic instructions may need several instants to complete. This is typically the case of the instruction `await`.

Communication and Synchronisation

The simplest way for threads to communicate is of course via shared variables. For example, a thread can set a boolean variable to indicate that a condition is true, and other threads can test the variable to know the status of the condition. This basic pattern works well when all the threads accessing the variable are linked to the same scheduler. Indeed, in this case atomicity of the accesses to the variable is guaranteed by the cooperativeness of the scheduler. A general way to protect a data from concurrent accesses is thus to associate it with a scheduler to which threads willing to access the data should first link to. Events are basically used by threads to avoid busy-waiting on conditions. An event is always associated with a scheduler (by default, the implicit scheduler), or with a set of synchronised schedulers. An event is either present or absent during

each instant. It is present if it is generated by the system (e.g. as a consequence of interactions with the environment) at the beginning of the instant, or if it is generated by one of the threads executed during the instant; it is absent otherwise. The presence or the absence of an event can change from an instant to another, but during one instant all threads always "see" the presence or absence of events in the same way, independently of the order in which the threads are scheduled. For this reason, we say that events are *broadcast*. Values can be associated with event generations; they are collected during the instant (`get_all_values` statement) and their collection becomes available as a list at the next instant.

Events are shared by synchronised schedulers: this is possible because synchronised schedulers share the same instants. Thus, if `s1` and `s2` are two synchronised schedulers, it is for example possible to wait for an event in `s1` and to generate the event in `s2`.

Memory Model

The global memory is divided in several parts:

1. A private memory for each thread. This memory is initialised when the thread is created, from the parameters and the local variables of the module from which the thread is created. The system statically checks that no other thread can have access to this private memory.

2. A private memory for each scheduler. This memory can only be accessed by the threads linked to the scheduler. The system statically checks that it is not accessible from unlinked threads, nor from threads linked to another scheduler.

Thus, there exists no global variable shared by distinct schedulers, and a variable that is shared by several threads belongs to the unique scheduler that controls these threads and cannot be accessed by other threads.

***

This terminates the overview of the model and the overview of the syntax of FunLoft. The next two chapters introduce the language in detail.

Chapter 3

Language Description

This chapter describes the syntax and the (informal) semantics of FunLoft. First, definitions are considered; then, values and expressions are introduced; finally, the chapter ends with the description of instructions.

3.1 Definitions

Programs are lists of definitions of variables, types, functions, extern definitions, modules, and schedulers. An executable program ia a program that contains the definition of a module named `main`.

3.1.1 Variables

A variable definition associates a name to the value of an expression. The syntax of variable definitions is:

```
let name = expression
```

The type of the defined variable is infered by the system from the expression.

3.1.2 Types

The format of a type definition is:

```
type name_1 = union_1
 ...
and name_k = union_k
```

Such a definition recursively defines k types, called *inductive types*, named `name_i` (note the absence of the keyword `rec` despite the recursive nature of the definition). Each union is a list of constructors, separated by the bar symbol "`|`".

A constructor is made of a constructor name with an optional tuple of types. A constructor without tuple is said of arity 0; otherwise the arity of the constructor is the length of the tuple. Constructor names are identifiers starting with a capital letter. For example, the type `direction_t` defines 4 constructors of arity 0, `TOP`, `LEFT`, `BOTTOM`, and `RIGHT`:

```
type direction_t = TOP | LEFT | BOTTOM | RIGHT
```

A constructor of arity $n > 0$ has the general form `C of t_1 *...* t_n` where `C` is a constructor name and the `t_i` are types. For example, lists of integers can be defined by:

```
type i_list = I_null | I_cons of int * i_list
```

This defines the type `i_list`, a value of which is either the constant `I_null` or a constructed value `I_cons (i,l)` made of an integer `i` of type `int` and of a value `l` of type `i_list` (e.g. `I_cons (0,I_null)`).

Parametrised types can be defined, in which parameters denote types and are identifiers prefixed by the backquote symbol. The parametrised type `‘a list` of lists made of values of type `‘a` is pre-defined in all programs:

```
type ‘a list = Nil_list | Cons_list of ‘a * ‘a list
```

In a program, all constructors should have distinct names. Thus, it is always possible for the system to infer the type of a constructed value from the constructor name.

3.1.3 Functions

A function definition has the general form:

```
let name_1 (param_list_1) = inst_1
...
and name_k (param_list_k) = inst_k
```

This definition recursively defines k functions, named `name_i` (as in type definitions, the keyword `rec` is absent). Each `param_list_i` is a list of comma-separated names which are the parameters of the function `name_i`. The function body `inst_i` is an instruction (see 3.3) which is executed, after parameter passing, each time the function `name_i` is called. Functions are “first-order” only: parameters and values returned cannot be themselves functions.

Here is for example the definition of the function which returns the length of a list of type `i_list`:

```
let length (l)  =
   match l with I_null -> 0 | I_cons (h,t) -> 1+length (t) end
```

The `match` instruction (defined in 3.2.2) is the way to de-construct values of inductive types, and to name their constituents.

Recursivity is controlled: it can only concern parameters of inductive types, and the size (in a sense to be precised later) of the parameters of a called function should always decrease (in the example, as `l` matches `I_cons (h,t)`, the size of `t` is less than the size of `l`).

3.1.4 External Definitions

External definitions declare variables or functions not defined in FunLoft but which can be nevertheless used in programs. The syntax for external variables is:

```
let name : type
```

The syntax for external functions is:

```
let name : type_1 *...* type_n -> type
```

For example, here are two definitions of extern variables, and one definition of an extern function:

```
let maxx       : int
let maxy       : int
let random_int : int  -> int
```

3.1.5 Schedulers

There are only a fixed number of running schedulers in an executable program; this is because scheduler definitions can only appear at the upper level of programs. Scheduler definitions have the form:

```
let name_1 = scheduler
...
and name_n = scheduler
```

This definition declares an *area* of `n` synchronised schedulers, named `name_i`, which are run asynchronously, but share the same instants. Schedulers are of the type `scheduler_t`. Events are shared by the schedulers belonging to the same area.

Note the difference with the following list made of single definitions which defines non-synchronised schedulers that share nothing, neither instants nor events:

```
let name_0 = scheduler
...
let name_n = scheduler
```

A scheduler, called the *implicit* scheduler, always exists in every program. The instance of the module `main` which starts the program execution is automatically linked to this scheduler.

Schedulers are run in parallel by dedicated native threads, which thus can be mapped on the cores of a multicore machine, when available. This is the way FunLoft basically addresses the issue of multicore architecture programming.

3.1.6 Modules

Modules are templates from which threads are created. A thread created from a module is called an *instance* of the module. A thread is always created in a scheduler to which it is linked and which is initially in charge of running it.

A module can have parameters which define corresponding parameters of its instances. Arguments provided when an instance is created are associated to these parameters. A module can also have local variables, new fresh instances of which are automatically created for each thread created from it. The syntax of modules is:

```
let module name (p_1,...,p_n) = body
```

where the `p_i` are the module parameters, and `body` is an instruction (see 3.3) defining the module body. As opposed to functions, modules cannot be recursively defined. As an example of module, consider the module named `trace` defined by:

```
let module trace (s)
  loop
    begin
      print_string (s);
      cooperate;
    end
```

This module has a parameter which is a character string (infered by the system because the parameter is given as argument to the predefined function `print_string`). The body is a loop (actually, an infinite one; see 3.3.2) which at each instant prints the parameter message and cooperates. The parameter `s` is actually a local variable which is passed at creation to the instances of `trace`.

A new instance of a module `m` is created using the syntax `thread m (...)`, defined in 3.2.2 which returns the new created thread. Threads are of the predefined type `thread_t`. The arguments given to `m` are passed to the created instance (as for functions, arguments are passed by value). For example, the following module creates two instances of the previous module `trace`:

```
let module two () =
   begin
     thread trace ("a");
     thread trace ("b");
   end
```

The output produced is a sequence of `a` and `b` such that there is at most 2 successive `a` and two successive `b` in it, for example "`ababab`...". Indeed, the way to produce 2 consecutive `a` is to chose the `a` thread as second during instant n and as first during instant $n+1$. But after having produced the second `a`, the only possible thread to be run is the one that produces `b`.

Several points are important:

- Threads are created in the scheduler of the executing thread. The system checks that no thread is created while unlinked.
- Threads are automatically started. This is a difference with Java in which threads have to be explicitly started.
- Threads are not immediately incorporated in the scheduler, but at the beginning of the next instant.

The entry point of an executable program is an instance created in the implicit scheduler of the `main` module. The global program does not terminate when the instance of `main` terminates; the programmer can force the program termination by calling the predefined function `quit` (see 3.2.3).

3.2 Values and Expressions

Evaluation of expressions are returning values. In what follows, to simplify, one often feels free not to distinguish between an expression and its value.

3.2.1 Basic Types

A basic value, which is a boolean (`true`, `false`), an integer value (e.g. `42`), a floating point value (e.g. `42.0`), a character (e.g. `'a`), a string (e.g. `"hello"`), or the unique value of the type `unit` (noted "`()`"). The system infers the type of basic values from their syntax. The expressions built on basic types are:

- The equality `e1=e2` and the inequality `e1<>e2`.
- `e1&&e2`, `e1||e2`, `not e`, where `e1,e2,e` are boolean expressions (as in C, `&&` denotes the boolean "and" , and `||` the boolean "or").
- `e1+e2`, `e1-e2`, `e1*e2`, `e1/e2`, `e1 mod e2`, `-e`, `e1>e2`, `e1<e2`, where `e1`, `e2`, and `e` are integer expressions.
- `e1+.e2`, `e1-.e2`, `e1*.e2`, `e1/.e2`, `-.e`, `e1> .e2`, `e1< .e2`, where `e1`, `e2`, and `e` are float expressions.

The division operation on numeric types (integer and float) is rather special in FunLoft: division by zero is trapped and thus crashes caused by such divisions are impossible. Actually, extremal values exist that are results of division by zero.

The operators on basic types are few in the present version of the language; more operators should certainly be defined in forthcoming versions (for example, $\leq$ and $\geq$).

3.2.2 Expressions

An expression is either:

- An identifier denoting a (local or global) variable, or a parameter of a function or of a module.
- A basic value or an expression on values of basic types.
- A value of an inductive type, build from a constructor (for example, `LEFT`, or `I_cons (3,l)`). The system infers the type of the value from the constructor name (possible because constructors always have distinct names).
- A global reference of the form `ref v` which defines a new memory location initially filled with the value of `v`. The system infers, from the use of the reference, the scheduler to which the reference belongs.
- A local reference of the form `local ref v` which defines a new memory location initially filled with `v`, and owned by the executing thread. The system checks that such a reference can only be accessed by its owner.
- A reading access to a reference, of the form `!r`, where `r` is an expression whose value is a reference; the access returns the value held by the reference.
- An assignment to a reference, of the form `r:=v` where `r` evaluates to a reference holding values of a type t and `v` evaluates to a value of t. The assignment is of type `unit` and fills the reference `r` with `v`.
- A function call of the form `f(e_1,...,e_n)`. The system checks from the definition of `f` that the parameters are correctly used.
- A creation of a thread. The creation of a new thread instance of a module `m` has the form `thread m (e_1,...,e_n)`. A thread has the type `thread_t`. The system checks that no thread creation can occur while unlinked. The new thread is created in the scheduler s to which the executing thread is linked. The new thread is started at the next instant of s, and it is the value of the creation instruction.
- A creation of an event. The creation of a new event has the form `event`. An event has the type `'a event_t`, the parameter being the type of the values generated with the event. The system infers which schedulers are concerned by the creation (there can be several, in the case of synchronised schedulers).

- The creation of an array has the form `ref[n] v`; it returns an array of `n` components, each of them being a reference initialised by `v`.
- The access to the `i`th component of an array `a` is noted `a[i]`. Note that this is always a reference or an array. FunLoft has a very special way to manage arrays as it basically considers them as *cyclic buffers*. Access to an element is indeed performed modulo the size of the array, which elmininates out-of-bound errors.
- The generation of an event has the form `generate e with v`; it is of type `unit`, and as a side-effect it generates the event `e` with the associated value `v`. The value can be omitted when it is the `unit` value `()`.
- The instruction `stop t` is an order to definitively stop the thread `t`. If `t` is unlinked, the order is lost; otherwise, it is transmitted to the scheduler to which `t` is linked. If `t` and the executing thread are both linked to the same scheduler, `t` will be stopped at the beginning of the next instant; otherwise, the stopping action is asynchronous. Similarly, the instructions `suspend t` and `resume t` are orders to suspend and resume `t`.
- The incrementation `r++`, where `r` is a reference on integers, is a short-hand for the assignment `r:=!r+1`; the decrementation `r--` is a short-hand for `r:=!r-1`.

3.2.3 Predefined Functions

Several functions are pre-defined:

- Conversion functions for numbers: `float2int` and `int2float` (`float2int` converts `float` to `int`; this is a general concention: function `x2y` converts `x` to `y`).
- `random_int` for producing a random integer, less than its argument.
- Printing functions for basic types: `print_int`, `print_float`, `print_string`, `print_char`, `print_bool`, `print_unit`.
- `flush` to flush the standard output and `print_newline` to print a new-line.
- `quit` to quit the application (with the same convention as the `exit` function of C: 0 for the normal case, and other values for abnormal cases).
- `dimension` which returns the dimension of an array given as argument.
- `fl_get_char` which returns the character typed on the keyboard. This is a blocking function: the system checks that it can only be used while unlinked.
- `fl_fopen`, `fl_fclose`, `fl_fread`, `fl_fnull` to manipulate file descriptors (of the predefined type `file_t`).

- Function for string manipulations: `length_string`, `concat_string`, and the two functions related to characters `string2char`, and `char2string`.
- Function `myself` which returns the executing thread, of type `thread_t`.

Note that the keywords `exit`, `abort`, and `kill` are trapped by the lexical analyser to forbid conflicts with the corresponding C functions.

3.3 Instructions

One makes the distinction between several kinds of instructions:

- General instructions can be used in functions and in modules.
- Infinite loops can appear in modules, even in an unlinked context.
- Non-atomic instructions whose execution may possibly last several instants. Non-atomic instructions can only appear in modules, but only in a linked context (instants are meaningless when unlinked).

3.3.1 General Instructions

General instructions can be used in functions and in modules, and are defined as follows:

- Expressions are instructions evaluated for their side-effects, and not for their value which is simply lost.
- The sequence of several instructions has the form `begin i_1;...;i_n end` (the last instruction `i_n` can be followed by a last meaningless semicolon).
- Execution of `repeat n do inst` first evaluates the value of `n`, then executes `n` times the instruction `inst`.
- Boolean test has the form `if cond then inst_1 else inst_2`. The `else` part can be omitted and replaced by the keyword `end`.
- Declaration of a variable has the form `let x = e in inst`. The expression `e` is first evaluated and its value is assigned to the variable `x` during the execution of `inst`.
- The match instruction has the form `match v with m_1 | ... | m_n end` when no default clause is given (then, all matching cases must be covered). It has the form `match v with m_1 | ... | default -> inst` when a default clause is given. The decomposed value `v` should be of an inductive type. Each match case `m_i` has the form `filter -> inst` with `filter` being a constructor name possibly followed by a list of new names denoting the constituents of the decomposed value.
- Execution of `return v` terminates the executing function or module and returns the value of `v`. The value `v` can be omitted when it is `()`.

3.3.2 Infinite Loops

The `repeat` loop of 3.3.1 is finite: there is no possibility with it to loop forever, and this is the reason why it can be used safely in function. In function definitions, recursion (which is controlled, as seen previously) is the other way to define cyclic treatments. The situation is different with modules, in which infinite loops can be used. The syntax is:

```
while cond do body
```

The value of the condition `cond` is only considered at the first instant, and each time the body terminates. If it is true, then the body is re-executed, and if it is false, the loop terminates. Note that `cond` is not evaluated at every instant, but only at the first instant and each time `body` terminates. For example, the following loop never terminates, despite the fact that the tested condition is false at some instants:

```
let b = ref true in
  while !b do
    begin
      b:=false;
      cooperate;
      print_string ("loop!");
      cooperate;
      b:=true;
    end
```

There is a simplified syntax for loops which never terminate:

```
loop body
```

is equivalent to

```
while true do body
```

Instantaneous loops are loops with a non-cooperating body. This is for example the case of `loop do print_string ("loop!")`. Instantaneous loops are admited only when unlinked; indeed, in this case, the executing thread, which is autonomous, cannot prevent other threads to execute. When linked, the system checks the absence of instantaneous loops which are obstacles to reactivity.

Let's now turn to the non-atomic instructions.

3.3.3 Cooperate

The basic non-atomic instruction is the `cooperate` instruction which returns the control back to the scheduler. When receiving the control after a `cooperate` instruction, the scheduler knows that the executing thread has finished its execution for the current instant, and thus that is is not necessary to give it

back the control another time during the instant. When the thread will receive the control in a future instant, the execution will resume in sequence from the `cooperate` instruction. Execution of a `cooperate` instruction thus needs at least two instants to complete (actually, it can take more than 2 instants if the executing thread is suspended after the cooperation).

3.3.4 Await

Execution of an `await` instruction suspends the executing thread until an event is generated. There is of course no waiting at all if the event is already present. Otherwise, the waiting can just take a portion of the current instant, if the awaited event is generated later in the same instant, by a thread scheduled later; the waiting can also last several instants, or even be infinite if the awaited event is never generated. The syntax is:

```
await evt
```

This instruction awaits the presence of the event `evt`.

There is a way to limit the time during which the executing thread is suspended waiting for an event. The limitation is a number of instants and the executing thread is resumed when the limit is reached. Of course, the waiting ends, as previously, as soon as the event is generated, before the limit is reached. An optional handler is executed when the timeout is exceeded. The syntax is:

```
await evt timeout num do handler
```

The expression `num` should be of type integer, and `handler` is an instruction. The handler part can be omitted when `handler` is simply the expression `()`. For example, the following code tests the presence of an event during the current instant:

```
let b = ref true in
  begin
    await e timeout 1 do b := false;
    if !b then print_string ("is present")
    else print_string ("was absent")
  end
```

Note that the message `was absent` is printed at the next instant in case of absence of `e`, while `is present` is immediately printed during the current instant in case of presence. This is a fundamental characteristics of the language: reaction to the absence of an event (here, the printing action) cannot be immediate, but is always postponed to the next instant.

3.3.5 Collecting Generated Values

There are two instruction to use the values associated with events. In both cases, the executing thread is suspended until the next instant.

The first instruction is the `get_all_values` instruction which collects all the values generated for an event and stores them in a list. The syntax is:

```
get_all_values evt in reflist
```

The execution of the executing thread is suspended until the next instant. During the current instant, all the values generated for `evt` are collected and put in a list referenced by `reflist` whose type is `'a list ref`.

The following module awaits an event with integer values and at the next instant it prints all the values generated for it:

```
let module print_all_values (evt) =
   let l = ref Nil_list in
      begin
         await evt;
         get_all_values evt in l;
         pr_list (!l);
      end

let pr_list (l) =
   match l with
        Nil_list -> ()
      | Cons_list (h,t) ->
          begin
             print_int (h);
             pr_list (t);
          end
   end
```

Note that the module always terminates at the instant that follows the one during which the event is generated. This should not be surprising: one must wait for the end of the current instant to be sure that all values have been effectively collected.

The second instruction has the form:

```
for_all_values evt with name -> call_back
```

where `call_back` is an expression in which `name` denotes the collected values. In contrast with the previous instruction, the values generated with `evt` are immediately processed, by evaluating `call_back` for each of them. The system checks that no event generation nor thread creation may be issued by `call_back`.

Let us consider the module:

```
let module print_immediate (evt) =
   begin
      await evt;
      for_all_values evt with x -> print_int (x)
   end
```

As opposed to `print_all_values`, values are immediately printed, one after the other, as they are generated. Note however that, like in `print_all_values`, execution terminates at the next instant only, to let the time to process all the generated values.

3.3.6 Join and Run

The `join body` instruction suspends the executing thread until complete termination of the instruction `body`; here, "complete" means that all the threads (transitively) created during the execution of `body` have also to be terminated. A `join` instruction terminates at the instant that follows the termination of the body; thus, a `join` instruction always takes at least one instant, even if its body terminates instantaneously. As example, the following loop is not instantaneous:

```
loop join 0
```

Note that there are actually two ways for a thread to terminate: either because nothing remains to be executed, or because the thread is stopped.

A module can run another one, using the `run` instruction. The calling thread suspends execution when encountering a `run` instruction. Then, an instance of the called module is created with the arguments provided. Finally, the calling thread is resumed when the instance of the called module terminates. Actually, the instruction:

```
run mod (...)
```

is equivalent to:

```
join thread mod (...)
```

3.3.7 Unlink

When unlinked, a thread becomes autonomous, which means that it executes independently of any scheduler, at its own pace. The syntax is:

```
unlink body
```

where `body` is an instruction which does not contain non-atomic instructions. When `body` terminates (if it does), the executing thread re-links to the initial scheduler and proceeds in sequence. An `unlink` instruction takes at least one instant to execute (even if the body terminates instantaneously). The system checks that the only references accessed in `body` are private to the executing thread.

For example, consider the following program which creates two threads instances of the same module `pr`:

```
let module pr (s) =
   unlink
     loop print_string (s)

let module main () =
   begin
      thread pr ("hello ");
      thread pr ("world!\n");
   end
```

The two lists of messages produced are merged in an unpredictable way in the output. Note that the loop in `pr` is instantaneous; this is not a problem as the thread is unlinked when it is executed. The granularity of each thread is, however, under the dependence of the OS, which can be problematic in some situations.

Unlinking is important for using standard blocking I/Os in a cooperative context. For example, the following module uses the `fl_getchar` function which blocks execution of the calling thread until a character is read on the standard input:

```
let module analyseInput () =
   unlink
      while ... do
         let c = fl_getchar () in ...
```

The first instruction unlinks the thread from the current scheduler. Then, the `fl_getchar` function can be safely called without any risk to block other threads. Note that an error is detected if `fl_getchar` is used when linked.

3.3.8 Link

When created, a thread is always linked to a scheduler (the main thread is automatically linked to the implicit scheduler). During execution, a thread can link to others schedulers, using the `link` instruction. The syntax is:

```
link sched do body
```

where `body` is the instruction to be run while linked to `sched`. The effect is to extract the executing thread from the current scheduler s and to add it to `sched` for the execution of `body`. When the execution of `body` terminates (if it does), the thread re-links to s and resumes execution at the next instant. Thus, a `link` instruction takes at least one instant to execute (even if the body terminates instantaneously). As example, the following loop is not instantaneous:

```
loop link s do 0
```

The `link` instruction can be seen as a restricted form of thread migration. The following module describes a bouncing behavior between two schedulers. The program prints "`Ping Pong`" forever:

```
let module play () =
   loop
      begin
         link sched1 do print_string ("Ping ");
         link sched2 do print_string ("Pong\n");
      end
```

Chapter 4

Static Analyses

This chapter describes the static checks that are performed in order to determine if a given program is correct. The focus is put on the checks that are specific to the language: checks to insure reactivity; checks to insure the basic separation of the memory; and ckecks to control the resources. Other checks are standard in functional languages, for example the type-checking of function parameters; among these checks, only the control on parametric types is considered here (as it is not completely standard).

4.1 Reactivity

Instantaneous Loops

To avoid instantaneous loops, the system checks that bodies of `loop` and `while` instructions cannot terminate instantaneously.

There are several instructions that are proved to never terminate instantaneously: `cooperate`, `get_all_values` and `for_all_values`, `run` and `join`, `link` and `unlink`, and finally `loop`. A sequence never terminates instantaneously if one of its components does so. A test never terminates instantaneously if its two branches do so. Other instructions (expressions `await`, `repeat` and `while`) can terminate instantaneously.

Note the difference between `loop i` and `while true do i`; the first is proved to never terminate instantaneously, but the second can (which is however false operationally, but the condition is not considered by the analysis which is conservative).

The following module is incorrect:

```
let module m () =
   loop
      begin
         await e;
         print_string ("received!\n");
```

```
      end
```

A way to make it correct is to add a `cooperate` statement at the end of the loop body.

The following module is correct because the collection of all the event values lasts one complete instant:

```
let module m () =
  let r = Nil_list ref in
   loop
      begin
         get_all_values e into r;
         print_string ("received!\n");
      end
```

For_all_values Instruction

No generation of events should be produced during evaluation of the callback expression of a `for_all_values` instruction; the risk is indeed to generate the event whose values are read, as in:

```
for_all_values e with x -> generate e with ()
```

The instruction would never terminate as the reaction to reading a value is to produce a new one.

Termination of Functions

Recursivity of function definitions is controlled: first, recursivity can only concern parameters of inductive types; second, in all sequences of calls that can be extracted from a recursive definition of functions, the size (in a sense to be precised later) of the calls appearing in the sequence should always decrease. Indeed, in this case, any function call is forced to terminate after a finite number of recursive calls.

As example, the definition of the following function `length` is correct:

```
let length (x) =
   match x with Cons_list (h,t) -> 1+length (t) | default -> 0
```

Indeed, there is only one sequence of calls, composed by the initial call `length(l)` and by the call `length(t)`. As `l` matches `I_cons(h,t)`, `t` is a strict sub-term of `l`. Thus, no infinite sequence of calls of `length` can exists, as the size of the parameter decreases at each call.

More precisely, the size of a parameter `p` is smaller than the size of a parameter `q` if `p` is a sub-term of `q`. For lists of parameters, one extends "lexicographically" the notion of size.

In the present version of FunLoft, recursivity can only concern parameters of inductive types. The following definition, in which recursivity concerns a parameter of integer type, is thus rejected:

```
let fact (n) =
   if n = 0 then 1 else n * fact (n-1)
```

The previous `fact` function is however expressible in FunLoft by:

```
let fact (n) =
  let res = ref 1 in
  let count = ref n in
  begin
    repeat n do begin
      res := !res * !i;
      i--
    end;
    !res
  end
```

Detection of termination of functions, as presently done in FunLoft, is a topics that should certainly be improved in further versions of the language. For example, in a sequence of calls leading from a function definition to a call of the same function, one could accept that some calls keep the parameters unchanged, provided there exists at least one call that strictly decreases the parameters size.

4.2 Memory Separation

The syntax of the language makes the distinction between local references (introduced by the keyword `local`) and the global ones (without `local`). However, at a logical level, one considers another distinction: *private* references are references that can only be accessed by one unique thread; other references, accessible by more than one thread, are said *public*.

One wants to identify local and private, and global and public. Moreover one wants to define an *ownership* relation that associates an owner to each reference, verifying the following *memory separation property*:

- A public reference can only be accessed by the threads linked to its owner (a scheduler).
- A private reference can only be accessed by its owner (a thread).

When it holds, the memory separation property makes interferences impossible: from the second part, no interference can involve a private reference; from the first part, no interference is possible between two threads linked to two distinct schedulers. As interferences are impossible by definition between threads linked to the same scheduler, there is no possibility at all to get interferences.

To express the separation property, a supplementary information called *status*, is added to types. This information indicates if a reference is private or public, and in the last case, the information precises the owner of the reference. The type system gives the private status to local references and tries to infer owners of global references.

Public References

The type system checks that the only public references accessed while linked to a scheduler s are those owned by s (CHECK1). Thus a public reference belonging to a scheduler cannot be accessed by a thread linked to another scheduler.

In the following example, the reference `r` is public. The system detects a contradiction from the definitions of the two modules `m1` and `m2`, as they imply that r should belong to the two schedulers:

```
let s1 = scheduler
let s2 = scheduler

let r = ref 0
let module m1 () = link s1 do r:=1
let module m2 () = link s2 do r:=1
let module m () =
    begin
       thread m1 ();
       thread m2 ();
    end
```

Private References

The type system checks (CHECK2) that only private references are accessed (read or written) while unlinked (that is, in the body of an `unlink` instruction). The following example is thus incorrect because the reference `r` is public (because it is not local) but accessed in the body of an `unlink` statement:

```
let module m0 () =
   let r = ref 0 in
      unlink r:=1
```

Declaring the reference as local turns the example to be correct:

```
let module m0 () =
   let r = local ref 0 in
      unlink r:=1
```

Transmission of Local References

The two previous checks are however not sufficient to insure the separation property because they do not forbid a thread to communicate one of its private reference to another thread that could thus access the reference. Two conditions are added: first, (CHECK3) a reference and its initialising value should have the same status. Second, (CHECK4) module parameters should always be public.

The rule of the assignment instruction guarantees that a local reference cannot be copied into a public reference. This property together with CHECK3

entails that a private reference cannot be transmitted to another thread through assignments.

CHECK4 forbids transmission of a private reference through module parameters. Without this check, a thread could have access to the private reference of another thread, as in:

```
let module m0 (x) =
   let r0 = local ref 0 in
      unlink
         begin
            r0:=1;
            x:=r0
         end

let module m1 () =
   let r1 = ref ref 0 in
      begin
         thread m0 (r1);
         !r1:=0
      end
```

Indeed, the local reference `r0` created in `m0` is written in `m1`, because it is the value of `r1`.

One can prove that the memory separation propery results from the previous four checks CHECK1-4. Thus, local and private may be considered as basically synonymous: a local reference can only be accessed by the thread in which it is declared, and thus can be considered as private to it. This forbids interferences between two unlinled threads, or interferences between a linked thread and an unlinked thread. Note that a private reference can always be accessed by it owner either linked or unlinked, without restriction. The impossibility of interferences between two threads linked to two distinct schedulers basically results from CHECK1.

4.3 Resource Control

Control on Thread Creations

A program in which the number of active threads is continuously increasing is erroneous as it will eventually run out of memory. Such programs should be rejected. This is for example the case with:

```
let module m () = loop cooperate

let module too_much_threads () =
  loop
     begin
```

```
        thread m ();
        cooperate
     end
```

A new instance of `m` is created at each instant and thus the number of simultaneously running threads is unbounded. The program has a memory leak, as the active threads never terminate (and thus cannot be collected) and therefore have to be stored somewhere. Moreover, as the number of cycles that has to be performed at each instant cannot be bound, the system is not reactive.

Threads however can be safely created cyclically at a given point in a loop when it can be proved that the threads previously created at that point are terminated before passing that point; these points correspond to the join primitive. The system thus checks that no thread can be created in an infinite loop statement (`while` and `loop`), except if the creation occurs in the body of a `join` instruction. For example, consider:

```
type tree = L of int | T of tree * tree

let map_on_leaf (t) =
   match t with L (n) -> thread leaf_processing (n)
          | T (t1,t2) ->
             begin map_on_leaf (t1); map_on_leaf (t2); end
   end

let module tree_processing () =
  let t = ref L (0) in
  loop
     begin
        t:=get_tree ();
        join map_on_leaf (!t);
     end
```

In this example, the loop stays bloqued on the join instruction until all the threads created for processing leaves are terminated. At each cycle, the number of created threads depends on the number of leaves of the tree t. However, as the size of t is bounded, this number is also bounded, like is the number of active threads in the system.

Size of Reference Values

Assignments of references of inductive types is controlled. Actually, only inductive types with a definition which is really recursive are considered; one says that these types are *infinite*. The previous type `list` is an exemple of infinite type: the size of values of this type is unbounded. The problem with an infinite type is to be sure that all values of this type are of bounded size.

References are stratified: there should be an order on reference accesses and no cycle is allowed to appear on these. Here is an incorrect function that violates stratification:

```
type nat = Z | S of nat
let nat_increm (n) =
   n := S(!n)
```

Indeed, a memory leak could appear if the function were accepted, as in:

```
let module memory_leak () =
   let r = ref Z in
   loop begin
     nat_increm (r);
     cooperate
   end
```

The size of the reference `r` is incremented at each instant, and thus a memory problem will eventually occur.

The system checks that, in all possible sequences of assignments, it is impossible for a reference to be assigned by a value whose evaluation needs to read the reference. Basically, the self-assignments `r:=!r` and `r:=f(!r)` are rejected. Self-assigments are also tracked along sequences of possible assignments; for example `begin r1:=!r2; r2:=!r1 end` is rejected.

At the implementation level, a *stratification* technique is used: the system tries to assign to each reference a stratification level (an integer), such that, in each assignment, the level of the left part is strictly less than the levels of all the references used in the right part. An error is found when this is not possible.

Only inductive types are concerned by stratification and the following function is thus correct:

```
let int_increm (i) = i:=!i+1
```

Note that there exist "correct" programs which are rejected. This is the case of the following module which is rejected despite the fact that the reference is reset at each instant, which actually forbids it to grow infinitely:

```
let module m (n) =
 loop begin
    n:=S(!n);
    cooperate;
    n:=Z;
  end
```

Size of Event Values

As references, events should also be stratified to avoid the risk of building infinite data, as in:

```
let module m () =
  let e = event in
  let r = ref Nil_list in
     loop
        begin
           generate e with Cons_list (0,!r);
           get_all_values e in r;
        end
```

This code is rejected as, on one side, the stratification level of `e` should be strictly less that the stratification level of `r`; but, on the other side, the inverse relation should also exists, which is contradictory.

4.4 Control on Parametric Types

If the system assigns to a variable an incomplete parametric reference, array, or event type (i.e. a type in which some parameters remains), then this variable should not be used in contexts where the parameters receive distinct types. One says that such a variable is *not generalisable*. For example, consider:

```
type ‘a cell_data = Undef | Cell of ‘a
let x = ref Undef
let f (v) =
 let z = !x in
     match z with Cell (c) -> x := Cell (v) | default -> ()
```

The type of `x` is not complete (one does not know what is `‘a`) and `x` is thus not generalisable. The function `f` has an effect which is to assign to `x` a value given in parameter. The following function `g` is erroneous as it first sets the parameter `‘a` of `x` to the boolean type, then to the integer type:

```
let g () =
   begin
      f (true);
      f (1);
   end
```

The problem is similar with events, and the following program is rejected (`e` is not generalisable, but used in two distinct contexts):

```
let e = event
let f1 (x) = generate e with x
let g () =
  begin
    f1 (true)
    f1 (1);
  end
```

Chapter 5

Programming Style

In Section 5.1, one considers the question: when does nondeterminism appears in FunLoft programs, and what are the techniques to control it? In Section 5.2, is considered the issue of protecting data from concurrent accesses. Section 5.3, considers division by zero and access to arrays.

5.1 Nondeterminism

There are several sources of determinism in FunLoft, as in most programming languages. As instance, a well-known cause of nondeterminism is the unspecified order in which function parameters are evaluated. For example, in the following code the value of `v` can be `1` or `2`, depending on the order of evaluation of the arguments of `f`:

```
let r = ref 0
let f (x,y) = !r
let v = f (r:=1,r:=2)
```

We shall not go in detail into these standard sources of nondeterminism, but instead consider the sources that are specific to FunLoft, and that are three in number:

- The order in which a scheduler executes the threads linked to it is not specified. This leads to *internal* nondeterminism.
- The way schedulers and unlinked threads are executed is not specified (they are run by native threads which are under the control of the operating system). This leads to *external* nondeterminism.
- The order in which the values generated with events are collected is left unspecified. This leads to *communicable* nondeterminism.

5.1.1 Internal Nondeterminism

One considers a program made of two threads linked to the same scheduler; one thread cyclically prints `a`'s and the other cyclically prints `b`'s. The printing module is defined by:

```
let module cyclic_print (s)
  loop
    begin
      print_string (s);
      cooperate;
    end
```

Note that the `cooperate` instruction is mandatory, as otherwise an instantaneous loop would be detected. The main module creates two threads, one for each kind of printing:

```
let module main () =
   begin
     thread cyclic_print ("a");
     thread cyclic_print ("b");
   end
```

The program is nondeterministic. In a notation where $+$ denotes the choice and $*$ the repetition, the set of possible outputs is $(ab + ba)^*$. In each output produced, there is at most 2 successive `a` and two successive `b`, as for example in "`baababba`...". This results from the following arguments. First, the two threads are started at the same instant, in the same scheduler. Second, each thread prints one and only one message at each instant. Third, the way to produce 2 consecutive `a`s is to execute the `a`-thread in second position during instant n, and in first position during instant $n + 1$. But after having produced the second `a`, the only possible thread to be run is the one that produces `b`.

Actually, an implementation will quasi-certainly always choose the same order of execution, producing for example "`abababa...`"; however, one must keep in mind that this is not implied by the language definition. In other words, the program is nondeterministic, while the implementation is.

Use of Events

Let us now turn to the question of getting a deterministic version of the previous program. A solution is to control the printing actions by means of events. One defines the controlled version of the previous `print` module by:

```
let module ctrl_cyclic_print (s,init,term)
  loop
    begin
      await init;
      print_string (s);
```

```
      cooperate;
      generate term;
    end
```

The main module creates two threads controlled by two triggering events, and generates the event corresponding to the a-thread:

```
let module main () =
  let trigger_b = event in
  let trigger_a = event in
   begin
     thread ctrl_cyclic_print ("a",trigger_a,trigger_b);
     thread ctrl_cyclic_print ("b",trigger_b,trigger_a);
     cooperate;
     generate trigger_a
   end
```

Then, the only possible output is "ababababa..." which is the only element of the set $(ab)^*$. Note the `cooperate` before the generation of `evt_b`; it is mandatory because the starting of the two threads only occurs at the second instant; without `cooperate`, event `trigger_a` would be generated at first instant and would thus be lost (and no printing would occur).

5.1.2 External Nondeterminism

Unlinked threads are a cause of nondeterminism. Let us for example consider the following module which cyclically prints a message while unlinked.

```
let module unlinked (s) =
  unlink
    loop
      begin
        print_string (s);
        flush ();
      end
```

Note that, as opposite to the previous module `trace`, the loop body does not contain a `cooperate` instruction, because the loop is executed while unlinked. The main module launches a linked thread to print `a` and an unlinked one to print `b`:

```
let module main () =
   begin
     thread cyclic_print ("a");
     thread unlinked ("b");
   end
```

The output is now an arbitrary mix of `a` and `b` (actually, it could be better for visualisation to replace one of `a` or `b` by a blank character). The set of outputs is $(a+b)^*$. The granularity of parallelism in this case only depends of the underlying operating system, and FunLoft gives no means to control it.

Several Schedulers

The presence of several schedulers is another source of nondeterminism. Let us define two schedulers and a printing thread in each of them:

```
let s1 = scheduler
let s2 = scheduler

let module main () =
   begin
     link s1 do thread cyclic_print ("a");
     link s2 do thread cyclic_print ("b");
   end
```

The output is an arbitrary mix of `a` and `b` element of $(a+b)^*$. As with unlinked threads, the granularity of parallelism only depends on the operating system.

Synchronised Schedulers However, it is possible to restrict the nondeterminism produced by the existence of several schedulers. The first means is to synchronise the schedulers:

```
let s1 = scheduler
and s2 = scheduler // s1 and s2 are synchronised
```

Now, the output is, as in 5.1.1, $(ab+ba)^*$.

It is even possible, using events as in 5.1.1, to get a complete determinism (events can be used here because the schedulers are synchronised). The program becomes:

```
let module main () =
  let trigger_b = event in
  let trigger_a = event in
   begin
     link s1 do
         thread ctrl_cyclic_print ("a",trigger_a,trigger_b);
     link s2 do
       begin
         thread ctrl_cyclic_print ("b",trigger_b,trigger_a);
         cooperate;
         generate trigger_a
       end
   end
```

Note that the initial triggering event can be generated in any of the two schedulers `s1` or `s2`, but not in the initial scheduler (an error is produced in this case).

Migration The use of shared events is impossible for schedulers which are not synchronised. In this case, migration is the solution.For example, let us define a `go_between` module which goes from one scheduler to the other, to drive two controlled printing threads:

```
let s1 = scheduler
let s2 = scheduler

let module go_between (inita,terma,initb,termb) =
   loop
     begin
       link s1 do begin generate inita; await terma; end;
       link s2 do begin generate initb; await termb; end;
     end
```

The program creates one printing thread in each scheduler, and an instance of `go_between` in the initial scheduler:

```
let module main () =
  let inita = event in
  let initb = event in
  let terma = event in
  let termb = event in
   begin
     link s1 do thread ctrl_cyclic_print ("a",inita,terma);
     link s2 do thread ctrl_cyclic_print ("b",initb,termb);
     thread go_between (inita,terma,initb,termb);
   end
```

The output is $(ab)^*$ despite the fact that the two schedulers are asynchronous. Note that one gets in this way a strong synchronisation of asynchronous schedulers.

5.1.3 Communicable Nondeterminism

The order in which the values generated with events are stored is not specified. For example, consider the following module which generates 2 values with the same event, and then processes them with a `for_all_values` instruction:

```
let module main () =
  let e = event in
    begin
      generate e with 1;
```

```
    generate e with 2;
    for_all_values e with v -> print_int (v);
  end
```

Two outputs are possible: `12` or `21`. There is no possibility to eliminate this form of nondeterminism, when `for_all_values` is used.

Nondeterminism is produced in the same way with the `get_all_values` instruction. However, in this case, the collected values are not immediately processed, but stored in a list. Thus, in this case nondeterminism only concerns the orders in which the elements are stored in the list.

5.2 Data Protection

The issue of data protection is fundamental in concurrent programming because of concurrent accesses to shared memory locations. Actually, data manipulated by one thread should be protected against accesses by other threads during a sequence of instructions that should be atomically executed. Data protection is thus basically a question of atomicity.

In FunLoft, very strong atomicity properties hold: on the first hand, execution of a linked thread between two cooperation points is proved to be logically atomic. Here, logically means that, while it is possible that instructions are physically interleaved, all goes on as if the execution is alone and isolated. On the other hand, unlinked threads cannot interfere, and thus also execute atomically.

In FunLoft all values, except references and arrays, are not mutable data which can thus be shared without problem. The sharing of the memory is reflected at syntax level by the sharing of references and of arrays.

The properties of the language leads to an approach which groups in the same scheduler logically related references that should be shared between several threads. In this approach, a thread that needs to access the data protected by a scheduler must first link to the scheduler in order to proceed. One thus gets a kind of monitor: the references can only be accessed by the threads linked to the scheduler which protect them from been accessed by other threads, either unlinked or linked to other schedulers. There are however several differences with monitors:

- Schedulers correspond to a kind of parallel monitor, as there can be several threads linked to it. A strong point here is that atomicity of accesses is automatically ensured. Another strong point is the possibility to use broadcast events as a communication mean for threads linked to the same scheduler (or to synchronised schedulers).

- A scheduler in FunLoft is run by a native thread; it is thus a "heavy" notion that should be used with caution. This is not the case of monitors that can be implemented in a lighter way.

Several questions are raised:

- How to deal with data processings that last accross several instants?
- How to proceed with manipulations that need to access several data regrouped on distinct schedulers?
- What to do if the use of a native thread for protecting a data is a too heavy solution?

We now try to answer these questions.

5.2.1 Multi-instant Processing

In settings based on preemptive scheduling, locks are used to protect shared resources from concurrent accesses. In FunLoft, nothing has to be done if accesses are instantaneous; indeed, in this case, accesses can be implemented by functions that are by definition run atomically. When accesses are not instantaneous, boolean variables can be used to protect shared data during several instants. This is basically not possible in preemptive settings because the sequence of instructions which test a variable and assign it a value is not atomic; this is why, locks have to be used. This is not the case in FunLoft which guarantees atomicity of test-and-set.

Let us consider a shared resource (let's say, a printer) that should be accessible by several threads, but used by only one at a time, and that perform long lasting actions. One defines a dedicated scheduler, a boolean reference to store the printer status, and an event to signal when the printer becomes available:

```
let sched_printer = scheduler
let in_use = ref false
let available = event
```

The `print` module first tests if the printer is in use; if the printer is in use, the `available` event is awaited. Otherwise, the `in_use` reference is set to true, and the printing action is performed. As this action takes several instants, it is implemented by a module. When the printing action is over (`run` instruction), the `in_use` flag is set to false and the `available` event is generated to awake waiting threads:

```
let module print (txt) =
  begin
    if !in_use then await available end;
    in_use := true;
    run print_action (txt);
    in_use := false;
    generate available;
  end
```

In this context, the instruction for printing has the form:

```
link sched_printer do run print (...)
```

Due to the signalisation through the event `available`, no busy-waiting is needed to wait for printer availability. Note that there is no risk that two users get the printer at the same time, because, in the body of `print`, the testing of `in_use` and its assignment by true are atomically executed.

Actually, the solution in FunLoft is rather similar to the one in a preemptive context like Posix: a boolean variable stands for a lock, and an event stands for a condition variable. The advantage is that this machinery must be installed only in case of long lasting actions, not for immediate ones (processed with functions) which are, by construction, automatically correct.

5.2.2 Multiple Data

In FunLoft, data should be mapped to schedulers, according to their proximity of use. Two data that are most of the time used together should belong to the same scheduler; conversely, two data that are independent can be mapped to distinct schedulers.

Actually, migration is the way to use several data mapped on distinct schedulers. Typically, to use two data, one on the scheduler `s1` and one on `s2`, a thread can migrate to these schedulers. The important point here is that *deadlock* situations can appear (no miracle...), exactly like in the standard preemptive context.

For example, let us define locks as boolean references, and the two basic primitives to use them as:

```
let module take_lock (lock) =
  begin
    while !lock do cooperate;
    lock := true
  end

let release_lock (lock) =
    lock := false
```

Note that the solution is very rough as it uses busy-waiting for waiting for lock to be free.

In the following code, `lock1` is taken first, then `lock2`:

```
link s1 do
  begin
    run take_lock (lock1);
    link s2 do
      begin
        run take_lock (lock2);
        ...
        release_lock (lock2);
```

```
      end;
    release_lock (lock1);
  end
```

Suppose that the same code in which the two locks are inverted is run in parallel with this one; then, a deadlock can appear, each piece of code taking one lock and being blocked on trying to take the other lock. Standard techniques for avoiding deadlocks in preemptive concurrency can be used here (as example, to always take locks in a fixed order).

The advantage of FunLoft, is again that in the simplest case (data mapped on the same scheduler and processed instantaneously), there is nothing special to do, and no deadlock can appear. Difficulties arise when processing may last several instants, or when data are mapped on distinct schedulers.

Note that there is the special case of data mapped on synchronised schedulers; in this case, theuse of events shared by the synchronised schedulers may help in solving the problem of accessing these data.

5.2.3 Protection by Threads

One may consider that it is not reasonable to use a scheduler for the protection of only one single data (or even for the protection of a set of related data). An argument for this position is that schedulers are implemented by native threads which may be too costly to use largely.

One may thus consider threads for the implementation of data protection. A way to do this would be to implement data as local references, and to let the managing thread react to events for data manipulation. This is an approach very close to the one proposed by object programming. To illustrate the approach, let us consider a data manager defined in the following way:

```
let module data_manager (order) =
  let data = local ref new_local_data () in
  let calls = ref Nil_list in
    loop
      begin
        await order;
        get_all_values order in calls;
        decode_list (!calls,!data);
      end
```

The function `new_local_data` returns a new data, of the type `data_t`. Let's suppose that two methods are defined to manipulate values of type `data_t`:

```
type method_t = Method1 | Method2
```

The function `decode_list` decodes values generated with the event `order`, by calling two functions `method1` and `method2`:

```
let decode (call,d) =
  match call with
     Method1 -> method1 (d)
   | Method2 -> method2 (d)
  end

let decode_list (list,d) =
  match list with Nil_list -> ()
    | Cons_list (call,tail) ->
        begin decode (call,d); decode_list (tail,d); end
  end
```

To call a method, a user just has to generate the `order` event with the method name as value, like in:

```
generate order with Method2
```

It would not be be very difficult to give parameters to methods, as in standard object programming. The point is that the system prohibits any access to the local data hold by a data manager, except the ones made by the manager. For example, an attempt by a method to transfer the data received in parameter into a global variable is detected as an error, and the program is rejected. Thus, data encapsulation can be realised in this way in FunLoft.

5.3 Run-time Errors

FunLoft adopts a rather extremist point of view concerning run-time errors issued from divisions by zero and out-of-bound accesses to arrays: these errors are trapped at run-time and thus never appear. The only run-time errors that remain are the ones that come when memory is exhausted.

5.3.1 Division by Zero

Integers in FunLoft correspond to the C type `long long int` (see the `limits.h` file for details). Division by zero is checked and trapped at run-time. The returned value of $n/0$ is `LLONG_MAX` if n is positive, and `LLONG_MIN` otherwise. Let's consider the following program:

```
let module main () =
  begin
    print_string ("1/0: LLONG_MAX = ");
    print_int (1/0);
    print_string ("\n-1/0: LLONG_MIN = ");
    print_int (-1/0);
    print_string ("\n1/0 + 1 = ");
    print_int (1/0+1);
    print_string ("\n");
  end
```

The output is:

```
1/0: LLONG_MAX = 9223372036854775807
-1/0: LLONG_MIN = -9223372036854775808
1/0 + 1 = -9223372036854775808
```

Note that adding one to `LLONG_MAX` evaluates to `LLONG_MIN`.

Division by zero can lead to a segmentation fault due to memory exhaustion, as when one defines a huge array by:

```
let a = ref[1/0] 0
```

Note that the existence of these run-time errors do not enter in contradiction with the properties issued from the static analyses; actually, the memory is bound, but the bound is so huge (greater than `LLONG_MAX`!) than no machine will probably be ever able to store such an array.

The present version of FunLoft identifies the `float` type with the one of C. As for integers, division by zero is trapped for float numbers. For example, let's consider the following code:

```
begin
  print_string ("\n1.0/.0.: HUGE_VAL = ");
  print_float (1.0/.0.);
  print_string ("\n-1.0/.0.: -HUGE_VAL = ");
  print_float (-.1.0/.0.);
end
```

The ouput is:

```
1.0/.0.: HUGE_VAL = inf
-1.0/.0.: -HUGE_VAL = -inf
```

5.3.2 Out of Bounds in Arrays

Arrays are cyclic in FunLoft. Thus, access to the ith element of an array a actually means $a[i$ `modulo` $n]$, where n is the size of a. For example, let's consider an array of 10 integers, and a function to fill it with consecutive values:

```
let a = ref [10] 0

let iota (a) =
  let i = ref 0 in
    repeat dimension (a) do
      begin
        a[!i] := !i;
        i++;
      end
```

Let's suppose we define a (probably bugged) function to print an array by:

```
let print (a) =
  let i = ref 1 in
    repeat dimension (a) do
      begin
        print_int (!a[!i]);
        print_string (" ");
        i++;
      end
```

Note the 1 which initialise the reference `i`. The sequence of calls:

```
begin
  iota (a);
  print (a);
end
```

produces the output:

```
1 2 3 4 5 6 7 8 9 0
```

One may say that the bug in `print` (initialisation of `i` by 1 instead of 0) remains masked as no error, nor exception is raised. This is true as long as one has in mind the standard representation of arrays. However, the argument is disputable when one remembers that arrays are cyclic buffers in FunLoft. The point is: are linear arrays preferable to cyclic arrays? The idea in FunLoft is that this is a matter of taste, and not a dogma. Anyway, some array processings are easier to write, for example the function that prints the array, starting from a given index:

```
let print_from (a,n) =
  let i = ref n in
    repeat dimension (a) do
      begin
        print_int (!a[!i]);
        print_string (" ");
        i++;
      end
```

Note that the following code (which in a standard setting can be seen as combining two major errors) is correct in FunLoft:

```
print_from (a,1/0);
```

To get in FunLoft the standard notion of an array, one must control all accesses to arrays, and define what to be done in case of a wrong access. Accesses should as well be controlled if one wants to code cyclic arrays in a standard language (as C).

Chapter 6

Basic Examples

One considers several basic examples which highlight some aspects of the language. Section 6.1 shows the benefit of a precise semantics. A notification-based communication mechanism is implemented in 6.2. Barriers are considered in 6.3. A reader/writer example is code in 6.4. A producer/consumer example is considered in section 6.5. Section 6.6 considers a reflex-game example, which is standard in reactive programming. Finally, section 6.7 considers a graphical example which shows how to buid a circle from two sine and cosine behaviors.

6.1 Mutual Stops

Let us consider a system made of two threads implementing two variants of a service, let's say, a fast one and a slow one. Two events are used to start the variants. After a variant is chosen, the other one should become unavailable; that is, each variant should stop the other variant. The coding of such an example is straightforward. First, event `start` is awaited; then, before serving (a message printed at each instant), the other variant is stopped; the code is:

```
let module variant (start,other,msg) =
   begin
      await start;
      stop !other;
      loop
        begin
           print_string (msg);
           cooperate;
        end
   end
```

Here is a possible use of the previous `variant` module:

```
let module main () =
```

```
let start_fast = event in
let start_slow = event in
let fast = ref null_thread in
let slow = ref null_thread in
  begin
     fast := thread variant (start_fast,slow,"fast\n");
     slow := thread variant (start_slow,fast,"slow\n");
     cooperate;
     generate  start_slow;
  end
```

First, two events and two threads are created, one for each variant; then, one variant is launched by generating the corresponding starting event (in this case, the slow variant is choosen). Note the two references, to store the threads, which are mandatory because each variant has to know the other one, and because no recursivity is possible in defining variables. Note also the cooperation after the creation of the two threads; without it, the starting event would be lost, as the threads become active only at the next instant.

The question is now: what happens if both `start_fast` and `start_slow` are simultaneously generated (logically, this should not appear but the question remains of what happens in this case)? The answer is clear and precise, according to the semantics: the two variants are executed during only one instant, and they both terminate at the next instant. Note that inserting a `cooperate` instruction just after the `stop` in `variant` would prevent both threads to start any servicing in this case.

Now, suppose that the same example is coded using standard Pthreads, replacing events by condition variables and `stop` by `pthread_cancel`. The resulting program is deeply non-deterministic. Actually, one of the two threads could cancel the other and run up to completion. But the situation where both threads cancel the other one is also possible in a multiprocessor context (where each thread is run by a distinct processor); however, in this case, both variants execute simultaneously during a while before cancellation, which produces unpredictable results.

Note that the mutual stopping exhibited in this example cannot be expressed in synchronous languages, (especially, in Esterel [16]) because it implies a *causality cycle*.

6.2 Wait/Notify

One considers thread synchronization using *conditions* which basically correspond to (simplified) *condition variables* of POSIX. A thread can *wait* for a condition to be set by another thread. The waiting thread is said to be *notified* when the condition is set. In a very first naive implementation, conditions are simply boolean references (initially false). To notify a condition means to set the reference to true, and to wait for the condition means to test it until notification.

```
let condition = ref false

let notify (cond) =
   cond := true

let module wait (cond) =
   begin
      while not !cond do cooperate;
      cond := false
   end
```

Note that no mutex is needed: because of the cooperative model, simple boolean shared variables are sufficient to implement atomic test-and-set operations. There is however a major drawback: the previous module `wait` performs busy-waiting and is thus wasting the CPU resource.

6.2.1 Avoiding Busy-Waiting

Events are the means to avoid busy-waiting. One now considers conditions as made of a boolean reference with an associated event:

```
type condition_t = Cond of bool ref * unit event_t
```

The notification function unsets the variable and generates the associated event:

```
let notify (cond) =
   match cond with Cond (done,go) ->
       begin done := false; generate go end
   end
```

The `wait` module awaits the event while the condition is not set:

```
let module wait (cond) =
   match cond with Cond (done,go) ->
     loop
       begin
           await go;
           if not !done then
              begin done := true; return end
           else
              cooperate
       end
   end
```

Note the `cooperate` instruction in the `else` branch of the test, to avoid an instantaneous loop. The first thread which receives the event `go` sets `done`, forbiding the other threads to proceed. Thus, one unique thread will proceed, the other continuing to wait for notification.

6.2.2 Notify All

Let us introduce, in addition, a way to notify *all* the threads waiting for a condition. First, one considers boolean events instead of unit events, as previously. The boolean value is used to distinguish between unique notification (`notify_one`) from multiple notification (`notify_all`). The type of conditions and the two notification functions are defined by:

```
type condition_t = Cond of bool ref * bool event_t

let notify_one (cond) =
   match cond with Cond (done,go) ->
       begin done := false; generate go with false end
   end

let notify_all (cond) =
   match cond with Cond (done,go) ->
       begin done := false; generate go with true end
   end
```

The `wait` module collects all the notifications received and decode them: if a multiple notification is received, then the module returns. It also returns if the notification is unique and no other thread has been choosen before (`done` is false). Otherwise, the waiting continues at the next instant:

```
let module wait (cond) =
   match cond with Cond (done,go) ->
     let all = ref false in
       loop
         begin
            await go;
            for_all_values go with x -> all := !all || x;
            if !all || not !done then
               begin done := true; return end
            end
         end
    end
```

Note that, by contrast with the previous implementation of `wait`, termination always occurs at the instant following the notification, because of the `for_all_values` instruction. When `notify_one` and `notify_all` are simultaneous (executed during the same instant), then `notify_one` has priority. This can be changed by replacing `&&` by `||` and by changing the initial value of `all`.

The solution proposed does not allow *targeted notification*, in which the notification is cibled to a particular thread. Actually, all threads waiting for the same condition are simultaneously awaken and all but one (with `notify_one`), or all of them (with `notify_all`) proceed. This can be inefficient when several threads are often simultaneously waiting for the same condition.

6.2.3 Targeted Notification

To be able to notify a specific thread, a dedicated event is associated to each waiting thread, used to trigger it. A condition now stores the events associated to the waiting threads; to notify a targeted thread, one just generates the associated event; to wait for a notification, a thread registers its associated event and then waits for it to be generated. The issue is thus the registration mechanism. Actually, one basically needs to store the set of waiting threads accross instants, with the possibility to add new elements at will. Basically, this contradicts stratification as the number of threads that are possibly waiting for a condition is not statically known. Thus, one cannot store the events associated to the waiting threads in a list as it is not possible in FunLoft to create a data whose size is unbounded.

Thus instead of a recursive data, one uses an array to store the threads waiting for a condition to be set. The price to pay is that one has to manage the cases where the number of waiting threads is greater than the array size. One chooses to consider arrays with a static size `max`; conditions are thus basically arrays of events:

```
type trigger_t = Undef | Event of unit event_t
type condition_t = Cond of int ref * trigger_t ref [max]
let new_condition () = Cond (ref 0,ref [max] Undef)
```

The function `register` searches for a free place in the condition array and, when possible, it fills it with a new triggering event:

```
let register (cond,go) =
 match cond with Cond (count,array) ->
   let i = ref 0 in
    repeat max do
      begin
         if !array[!i] = Undef then
             begin
                array[!i] := Event (go);
                count++;
                return
             end
         end;
         i++;
      end
 end
```

The `trigger` function generates the triggering event hold at a defined place of the condition array:

```
let trigger (cond,i) =
 match cond with Cond (count,array) ->
```

```
  let t = !array[i] in
   match t with Event (e) ->
      begin
          generate e;
          array[i] := Undef;
          count--;
      end
 | default -> ()
 end
```

The notify/wait primitives can now be defined. The notify function search for a defined place in the condition array, and if there is one, it triggers the corresponding waiting thread:

```
let notify (cond) =
 match cond with Cond (count,array) ->
   let i = ref 0 in
    repeat max do
      begin
         if !array[!i] <> Undef then
             begin
                fire (cond,!i);
                return
             end
         end;
         i++;
      end
 end
```

The module `wait` uses the counter of the condition to register a new triggering event in it; if it succeeds, then the module awaits the event. Otherwise, the module simply cooperates:

```
let module wait (cond) =
 match cond with Cond (count,array) ->
   begin
      while !count = max do cooperate;
      let go = event in
         begin
            register (cond,go);
            await go
         end
   end
 end
```

Note that the way the event array is managed is arbitrary and can be changed quite simply; this is of course a question which is inherent to the `notify` primitive: how is the notified thread chosen? Either which thread is notified is left

unspecified, which introduces some nondeterminism, or the way threads are chosen is specified, which can be felt as an over-specification. This is actually an issue which is to be solved by application programmers according to their needs.

The notification of a thread does not awake the other threads waiting for the condition and registered in the condition, because these are actually waiting on distinct events. When the number of waiting threads exceeds the array size, only the threads that have not suceeded to register are obliged to busy wait; the busy waiting of a thread lasts as long as it fail to register.

6.3 Synchronisation Barriers

A thread reaching a *synchronisation barrier* blocks until a certain number of other threads also reach the same barrier. When the threshold is reached, all the threads proceed together. The type of synchronisation barriers is defined by:

```
type sync_barrier_t =
    Sync of int ref      // counter
          * int          // threshold
          * unit event_t // proceed event
```

In `Sync (r,t,e)`, `r` is the counter of threads having already reached the synchronisation barrier, `t` is the threshold, and `e` is the proceed event. To reach a synchronisation barrier `b`, a thread executes `run sync (b)`, with the module `sync` defined by:

```
let module sync (barrier) =
   match barrier with Sync (c,t,e) ->
       if !c = (t - 1) then
          begin c++; await e end
       else
          begin c := 0; generate e end
   end
```

If the threshold is reached, then the counter is reset to 0 and the proceed event is generated; thus, all the waiting threads immediately proceed. If the threshold is not reached, then the counter is incremented and the executing thread awaits the proceed event to continue.

6.4 Readers/Writers

One considers several threads that are reading and writing a shared resource (this example is considered in the book *General Programming Concepts: Writing and Debugging Programs*, belonging to the AIX system documentation (2nd edition, 1999)). The writers have priority over readers. Several readers can

simultaneously read the resource while a writer must have exclusive access to it (no other writer nor reader can access it) while writing. One adopts the terminology of locks and note `rwlock_t` the type of control structures:

```
type rwlock_t = Rwlock of
   int ref      * // lock_count
   int ref      * // waiting_writers
   unit event_t * // read_go
   unit event_t   // write_go
```

The convention for `lock_count` is the following: 0 means that the lock is held by nobody; when held by a writer, `lock_count` has value -1; when positive, `lock_count` is the number of readers currently reading.

6.4.1 Writer

In order to proceed, a writer must first run the module `write_lock`:

```
let module write_lock (rw) =
  match rw with Rwlock (lock_count,waiting_writers,read_go,write_go) ->
    begin
       waiting_writers++;
       while !lock_count <> 0 do
          begin
             await write_go;
             cooperate;
          end;
       waiting_writers--;
       lock_count := -1;
     end
  end
```

When the writing action is finished, the writer must call the following `write_unlock` function:

```
let write_unlock (rw) =
  match rw with Rwlock (lock_count,waiting_writers,read_go,write_go) ->
    begin
      lock_count := 0;
      if !waiting_writers = 0 then
         generate read_go
      else
         generate write_go
    end
  end
```

6.4.2 Readers

A reader can proceed if no writer is currently writing and if there is no waiting writer:

```
Let module read_lock (rw) =
  match rw with Rwlock (lock_count,waiting_writers,read_go,write_go) ->
    begin
       while (!lock_count < 0) || (!waiting_writers > 0) do
          begin await read_go; cooperate; end;
       lock_count++;
    end
  end
```

After reading, a reader should call the function `read_unlock`:

```
let read_unlock (rw) =
  match rw with Rwlock (lock_count,waiting_writers,read_go,write_go) ->
    begin
      lock_count--;
      if !lock_count = 0 then generate write_go else ()
    end
  end
```

6.5 Producers/Consumers

One considers the standard example of producers and consumers. First, a variant with only one scheduler is considered. Then, the case of two schedulers is described.

6.5.1 Unique area

One implements the simplest form of a producers/consumers system where several threads are processing values placed in a shared buffer. Each thread gets a value from the buffer, makes some processing, and then put the result back in the buffer. To simplify, one considers that values are integers that are decremented each time they are processed; moreover, processing terminates when 0 is reached. First, a buffer implemented as an array, and an event are defined:

```
let buffer = ref [max_size] 0
let new_input = event
```

The processing module cyclically gets a value from the buffer, tests if it is zero, and if not, processes the value. When processing is finished, the value decremented by one is put back in the buffer. The event `new_input` is used to avoid busy-waiting while the buffer is empty

```
let module process () =
  let r = ref 0 in
  loop begin
     while length (buffer) = 0 do
       begin
          await new_input;
          cooperate;
       end;
     r := get (buffer);
     if !r > 0 then
       begin
         <processing>
         r--;
         put (buffer,!r);
         generate new_input;
       end
     end;
     cooperate;
  end
```

Note that, due to the language definition, the shared buffer is protected from accesses by unlinked threads or by threads linked to an other scheduler.

6.5.2 Two areas

One now considers a situation where there are two buffers `in_buffer` and `out_buffer`, and a pool of threads that take data from `in_buffer`, process them, and then put results in `out_buffer`. A distinct scheduler is associated to each buffer.

```
let in_buffer = ref [max_size] 0
let out_buffer = ref [max_size] 0
let in_sched = scheduler
let out_sched = scheduler
let new_input = event
let new_output = event
```

The module `process` gets values from `in_buffer`, avoiding busy-waiting by using the event `new_input`. A new thread instance of `process_value` is run for each value.

```
let module process ()
  loop begin
     if length (in_buffer) > 0 then
        run process_value (get (in_buffer))
     else
        begin
```

```
            await new_input;
            cooperate
         end
   end
```

The `process_value` module processes its parameter and then links to the scheduler `out_sched` to deliver the result in `out`. At delivery, the event `new_output` is generated to awake the threads waiting for `out`:

```
let module process_value (v)
   begin
      <processing the value>
      link out_sched do
         begin
            put (out_buffer,v);
            generate new_output;
         end
   end
```

This code shows a way, based on schedulers, to manage shared data. The use of standard locks is actually replaced by linking operations. Of course, locks still exist in the implementation and are used by linking operations, but they are totally masked to the programmer who can thus reason in a more abstract way, in terms of linking actions, and not in terms of low-level lock primitives.

6.6 Reflex Game

In this section, one considers the example of a little game for measuring the reactivity of users. This example, issued from Esterel [15], shows how FunLoft can be used for basic reactive programming.

The purpose of the game is to measure the reflexes of the user. Four keys are used: the `c` key means "put a coin"; the `r` key means "I'm ready"; the `e` key means "end the measure", and the `q` key means "quit the game". After putting a coin, the user signals the game that he[1] is ready; then, he waits for the `GO!` prompt; the measure starts after the prompt, and it lasts until the user ends the measure. After a series of measures, the game outputs the average score of the user. The game is over when an error situation is encountered. There are actually two such situations: when the player takes too much time to press a key (the player has abandonned); or when `e` is pressed before `GO!` (this is considered as a cheating attempt).

Variables

Several definitions are first introduced:

[1] Only a male can play to such a stupid game...

- Four external integer variables: `pause_length`, `limit_time`, `total_time`, and `measure_number`.
- Five events: `coin_evt` is generated when `c` is pressed; `ready_evt` is generated when `r` is pressed; `end_evt` is generated when `e` is pressed; `tilt_evt` is generated by the game when an error is detected; the value of `display_evt` is the measured time to be printed.
- A reference `total_time` which holds the cumulative time measured.
- A reference on a thread which holds the running party, to be able to kill it when needed (for example, when a cheating attempt is detected).

```
let pause_length : int
let limit_time : int
let measure_number : int
let coin_evt = event
let ready_evt = event
let end_evt = event
let display_evt = event
let tilt_evt = event
let total_time = ref 0
let party = ref null_thread
```

Game Phases

Phase 1

The first phase consists in waiting the notification that the user is ready. The waiting lasts at most `limit_time` instants, and, if the timeout is reached, an error is produced and `tilt_evt` is generated. During the waiting, mistakes are signaled by a beep sound: a mistake here means that the user presses the `e` key instead of the `r` key. Beeps are produced by the following module `beep_on`:

```
let module beep_on (e) =
   loop begin
      await e;
      print_string ("\07");
      cooperate;
    end
```

The first phase starts by creating an instance of `beep_on` which reacts on `end_evt`. Then, `ready_evt` is awaited during at most `limit_time` instants. If the timeout is reached, then `tilt_evt` is generated. In both cases, the previously created instance of `beep_on` is stopped.

```
let module phase1 () =
  let beeper = thread beep_on (end_evt) in
```

```
   begin
      print_string ("press r when ready\r\n");
      await ready_evt timeout limit_time do generate tilt_evt;
      stop beeper
   end
```

Phase 2

In phase 2, a prompt message is output after a random number of instants and then a new measure starts. Cheating attempts are detected: the player is cheating if he tries to anticipate the prompt by pressing `e` in advance. In this case, an error is detected and the party is over. The module `detector` detects the presence of `end_evt`, and then generates `tilt_evt` and stops its parameter:

```
let module cheat_detector (t) =
   begin
      await end_evt;
      stop t;
      generate tilt_evt;
   end
```

The module `phase2` first creates a cheating detector with itself as parameter, and, then, waits for a random number of instants (returned by the function `random_int`) before printing the prompt message. At that moment, the cheating detector is stopped.

```
let module phase2 () =
   let me = myself () in
   let detector = thread cheat_detector (me) in
      begin
         print_string ("wait...\r\n");
         repeat limit_time + random_int (limit_time) do cooperate;
         stop detector;
      end
```

Phase 3

Phase 3 consists in measuring the number of instants taken by the player to press `e`. An abandon (`e` is not pressed during `limit_time` instants) terminates the party. Moreover, mistakes (here, pressing `r` instead of `e`) are detected and signaled. Finally, the value measured is associated to `display_evt` to be printed.

An auxiliary module `increm` for counting instants is first defined which, once started, increments at each instant a reference passed as parameter:

```
let module increm (counter) =
   loop begin
      counter++;
```

```
      cooperate
   end
```

The phase3 module is defined by:

```
let module phase3 () =
  let count = ref 0 in
  let counter = thread increm (count) in
  let beeper = thread beep_on (ready_evt) in
    begin
       print_string ("GO!\r\n");
       await end_evt timeout limit_time do generate tilt_evt;
       stop counter;
       stop beeper;
       count := !count/10;
       total_time := !total_time + !count;
       generate display_evt with !count;
   end
```

Reflex Game Module

The final_display module waits during pause_length instants before generating display_evt with a value which is the average of the measured times:

```
let module final_display () =
   begin
      repeat pause_length do cooperate;
      print_string ("**** final ");
      generate display_evt with !total_time / measure_number
   end
```

A measure consists in running in sequence the three previous phases. The module list_of_measures executes measure_number measures and finally runs an instance of final_display:

```
let module list_of_measures () =
   begin
     repeat measure_number do
        begin
           run phase1 ();
           run phase2 ();
           run phase3 ();
        end;
     run final_display ();
   end
```

The one_game module waits for c and then executes a list of measures. The thread running the list of measures is stored in the global reference party, allowing one to stop it in case of errors:

```
let module one_game () =
  begin
     print_string ("Press c to start, q to stop.\r\n");
     total_time := 0;
     await coin_evt;
     party := thread list_of_measures ();
     cooperate;
     print_string ("game over.\r\n");
  end
```

Finally, one defines the `reflex_game` module which cyclically runs the module `one_game`:

```
let module reflex_game () =
   begin
      print_string ("A reflex game ...\r\n");
      print_string (
        "You must press ‘e’ as fast as possible after GO!.\r\n");
      loop run one_game ();
   end
```

Input/Output

Input

The input of the program is produced from the keys pressed. The keys are returned by the function `fl_getchar` which is not cooperative, and thus can be called only while unlinked. Two extern functions are called at the very start and at the end of the game:

```
let the_beginning : unit -> unit
let the_end : unit -> unit

let module analyse_input () =
  let c = local ref ' ' in
   begin
      the_beginning ();
      loop
         begin
            unlink c := fl_getchar ();
            if !c = 'c' then generate coin_evt
            else if !c = 'r' then generate ready_evt
            else if !c = 'e' then generate end_evt
            else if !c = 'q' then the_end ()
            else ();
            cooperate;
        end
    end
```

Output

The results of measures are generated as values of `display_evt`. The way to get them is to use the `for_all_values` instruction. The module `analyse_display` is:

```
let module analyse_display () =
  let res = ref 0 in
    loop begin
      await display_evt;
      for_all_values display_evt with v -> res := v;
      print_string ("score: ");
      print_int (!res);
      print_string ("\r\n");
      cooperate;
    end
```

A message with a beep must be issued when `tilt_evt` is generated; moreover, the party is over in this case:

```
let module analyse_tilt () =
   loop begin
      await tilt_evt;
      print_string ("\07TILT!!\r\n");
      stop !party;
      cooperate;
   end
```

Main Module

To end the description of the game, one defines the main module which launches the four threads needed:

```
let module main () =
   begin
      thread analyse_display ();
      thread analyse_tilt ();
      thread analyse_input ();
      thread reflex_game ();
   end
```

Execution Environment

One now considers the executing environment needed to use the reflex game. The 3 variables `limit_time`, `measure_number`, and `pause_length` have type `value`. The type `value` is defined in the interface file `val.h`. The 3 variables are set by the function `extern_constants` which is automatically called by the

system at the beginning of each run. The two functions `the_beginning` and `the_end` are defined to put the terminal in raw mode (to be able to directly get the key pressed by the player), and to restore the initial mode and terminate the game. The code of the C file is:

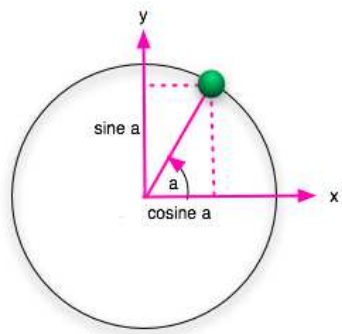
```
#include <stdio.h>
#include <stdlib.h>
#include "val.h"

value limit_time;
value measure_number;
value pause_length;

static int delay = 1000000;

void extern_constants (void)
{
  limit_time = int2val (delay);
  pause_length = int2val (delay/2);
  measure_number = int2val (4);
}

void the_beginning (void)
{
   system ("stty raw -echo");
}

void the_end (void)
{
   printf ("It's more fun to compete ...\r\n");
   system ("stty -raw echo");
   exit (0);
}
```

The integer values of C are transformed into elements of the type `value` by the `int2val` function, which belongs to the system library. Function `extern_constants` is by default defined with an empty body, in the system library; it is replaced by a user-defined function, when provided.

Note that, in order to obtain a playable system, the value of `delay` (which defines `pause_length` and `limit_time`) has to be adapted to the executing platform (the present value corresponds to a 2.33 GHz Intel Core 2 Duo machine).

6.7 Sine + Cosine = Circle

Let us consider the so-called "trigonometric circle" (whose radius is equal to 1):



For a point on the circle, defining an angle a with the x-axis, the projection on the x-axis is equal to $cosine(a)$ and the projection on the y-axis is equal to $sine(a)$.

Let us now consider the example of a graphical applet made of a ball animated by two threads: one animates the ball according to the sine function, while the other animates it according to the cosine function. A circle should be observed when the two behaviors are run together.

The type of balls is defined by:

```
type ball_t =
   Ball of
      int ref *     // x coord
      int ref *     // y coord
      float ref *   // x angle
      float ref *   // y angle
      int *         // x center
      int           // y center

let new_ball_at (x,y) =
   Ball (ref 0,ref 0,ref 0.0,ref 0.0,x,y)
```

Several variables and extern functions are defined:

```
// Dimension of the applet and size of particles.
let maxx = get_maxx ()
let maxy = get_maxy ()
let size = get_size ()

let step = 0.1
let radius = 100.0

let sine       : float -> float
let cosine     : float -> float
```

At each instant, the `x_behavior` makes a ball move in the x dimension. First, the angle in x is incremented, and then the x coordinate is computed as the cosine of the angle:

```
let move_x (me) =
  match me with Ball (x,_,anglex,_,_,_) ->
    begin
      anglex := !anglex +. step;
      x := float2int (radius *. cosine (!anglex))
     end
  end

let module x_behavior (me) =
  loop
    begin
      move_x (me);
      cooperate;
    end
```

In a similar way, at each instant, the `y_behavior` makes a ball move in the y dimension (with function cosine replaced by sine):

```
let move_y (me) =
  match me with Ball (_,y,_,angley,_,_) ->
    begin
      angley := !angley +. step;
      y := float2int (radius *. sine (!angley))
    end
  end

let module y_behavior (me) =
  loop
    begin
      move_y (me);
      cooperate;
    end
```

The `draw_behavior` forces the ball to be drawn on the screen at each instant:

```
let module draw_behavior (me) =
   loop
     begin
        match me with Ball (x,y,_,_,centerx,centery) ->
           draw_ball (!x+centerx,!y+centery,size,WHITE)
        end;
        cooperate;
     end
```

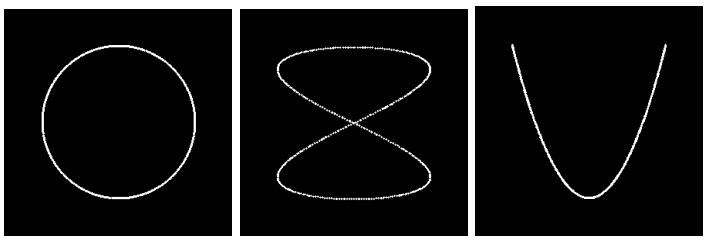
A circle is obtained by creating a new ball, and by launching 3 associated threads: one for moving the ball according to the x coordinate, one for moving it according to the y coordinate, and one for drawing it:

```
let module shape (x,y) =
  let b = new_ball_at (x,y) in
    begin
       thread x_behavior (b);
       thread y_behavior (b);
       thread draw_behavior (b);
    end
```

The main module creates a graphic display and a shape in the middle of it:

```
let module main () =
  begin
    thread remanent_graphics (maxx,maxy);
    thread shape (maxx/2,maxy/2)
  end
```

On the left part of the following figure is the circle obtained; on the middle part is the Lissajous shape obtained by adding a supplementary instance of x_behavior; on the right part is the shape obtained by adding to the circle a supplementary instance of y_behavior:



This example shows a direct use of the thread synchronisation related to instants. Note that no lock is needed to protect the shared ball from concurrent accesses.

Chapter 7

Cellular Automata

Cellular automata (CA) are used in various simulation contexts, for example, physical simulations, fire propagation, or artificial life. These simulations basically consider large numbers of small-sized identical components, called *cells*, with local interactions and a global synchronized evolution. Conceptually, evolution of CA is decomposed into couples of steps: during the first step, cells get the states of their neighbors and during the second step they change their own state according to information obtained from previous step. Usually, CA are coded as sequential programs basically made of a single main loop which considers all cells in turn.

7.1 Game of Life

In this section, one considers a very basic, although well-known, sort of CA, called *Game Of Life* (GOL). In GOL, cells are either alive or dead, depending of their neighbours[1]. The living status of a cells depends on the number of its neighbours which are alive. The changes of living status is defined by:

- A dead cell becomes alive when it has exactly 3 living neighbours.
- A living cell becomes dead when the number of its living neighbours is less than 2 or greater than 3.

7.1.1 Cells

The implementation of cells is based on the following points:

- Each cell is implemented by a thread, with an associated event and a boolean which indicates if the cell is alive or dead.

[1]The neighbourhood of a cell is constituted by its 8 adjacent cells; this neighborhood is called the *Moore* neighborhood.

- To activate one of its neighbors, a cell just generates the activation event of the neighbor.
- At each instant, a cell activates all its neighbors, then it collects all the activations from its neighborhood, and finally, it updates its state according to the received information.

In many CA, there is the distinction between *active* and *quiescent* cells. Quiescent cells actually always remain quiescent while their neighborhood is only composed of quiescent cells (empty neighborhood). Implementations should consider this and avoid any processing of quiescent cells. Actually, a cell cyclically performs the following sequence of actions:

1. if the cell is quiescent, it falls asleep, awaiting an activation from its neighborhood, and then it proceeds to action 2;
2. the cell activates all its neighbours;
3. the cell collects the number of activations comming from its neighbours; actually, only the neighbours which are alive are activating the cell;
4. the cell changes its state according to the number of activations received.

In this way, a quiescent cell does not perform any "busy-waiting" while its neighborhood remains empty. Note that, at each instant, only the living cells and their neighbours are executed; the other cells being quiescent, are just awaiting to be activated.

The data associated with each cell is made of an activation event, used by its neighbours to communicate their presence, and of a boolean defining the cell state (alive, true; dead, false):

```
type cell_data_t =
   Undef | CellData of unit event_t * bool ref
```

7.1.2 CA Space

Cellular automata considered here are 2-dimensions matrix of cell data; the matrix dimensions are given by the two constants `maxx` and `maxy`:

```
let ca_space = ref [maxx] ref [maxy] Undef
```

Initially, the matrix is filled with the `Undef` data; the initial configuration of the CA will be considered later.

The data associated to a cell can be accessed through its coordinates using the function:

```
let dataref_at (x,y) = (!ca_space [x]) [y]
```

One choses a standard toric geometry for the CA space. Note that the toric geometry is very natural in FunLoft, because it directly results from the definition of arrays as cyclic buffers. Basically, nothing has to be said to obtain this geometry.

The auxiliary function `event_at` returns the activation event embedded in a cell data. Using it, the `awake_neighbours` function activates the 8 neighbours of a cell by generating their associated events:

```
let event_at (x,y) =
  let d = !dataref_at (x,y) in
     match d with CellData (e,a) -> e | default -> event

let awake_neighbours (x,y) =
 begin
   generate event_at (x-1,y);
   generate event_at (x+1,y);
   generate event_at (x,y-1);
   generate event_at (x,y+1);
   generate event_at (x+1,y-1);
   generate event_at (x-1,y-1);
   generate event_at (x+1,y+1);
   generate event_at (x-1,y+1);
 end
```

7.1.3 Living Strategy

The living strategy for GOL is coded by:

```
let gol_strategy (living,neighbour) =
   if not !living && neighbour = 3 then
      living := true
   else if !living && neighbour <> 2 && neighbour <> 3 then
      living := false
   else ()
```

Living cells are displayed in yellow (dead cells are not desplayed at all):

```
let gol_display (x,y,living) =
   if living then
      draw_rectangle (x*zoom,y*zoom,zoom,zoom,YELLOW)
   end
```

Let us turn now to the behavior of cells, which is the following: first, the cell is displayed; if it is dead, then the triggering event is awaited (in which case the behavior does not consume any CPU resource while it remains dead); otherwise, the cell neighbours are awaken. Second, the number of living neighbours is collected and counted. Finally (at the next instant), the function `gol_strategy` is called to set the cell state:

```
let module cell (x,y,d) =
  match d with CellData (trigger,living) ->
    let count = ref 0 in
      loop
        begin
          gol_display (x,y,!living);
          if not !living then
            await trigger
          else
            awake_neighbours (x,y);
          count := 0;
          for_all_values trigger with _ -> count++;
          gol_strategy (living,!count)
        end
  | default -> ()
```

7.1.4 Creation of CA

The `create_cell` function creates a new cell, initially dead, stores its data in the CA space, and launches a thread to run the cell behaviour:

```
let create_cell (x,y) =
 let d = CellData (event,ref false) in
   begin
      dataref_at (x,y) := d;
      thread cell (x,y,d);
   end
```

The function `create_ca` fills the CA space with new cells:

```
let create_ca () =
  let x = ref 0 in
  let y = ref 0 in
  repeat maxy do
    begin
      x:=0;
      repeat maxx do
        begin
          create_cell (!x,!y);
          x++;
        end;
        y++;
    end
```

7.1.5 Graphics

The graphical level used here is the SDL Library[9] which gives a simple and powerful abstraction of graphics. The module `graphics` drives the graphical li-

brary: first, the library is initialized by a call to the function `initialise_graphics`, then the function `update_display` is called at each instant:

```
let module graphics (maxx,maxy,color) =
  let zoom = get_zoom () in
    begin
      initialise_graphics ();
      loop
        begin
          update_display ();
          draw_rectangle (0,0,maxx*zoom,maxy*zoom,color);
          cooperate;
        end
    end
```

7.1.6 Initial Configuration

The `fire` function forces the cell designed by its parameters to be alive:

```
let fire (x,y) =
  let data = !dataref_at (x,y) in
    match data with
        CellData (e,a) -> a := true | default -> ()
```

The function `r_pentomino` creates the well-known 5 cells shape called *pentomino*:

```
let r_pentomino (x,y) =
  begin
    fire (x,y);
    fire (x+1,y);
    fire (x+1,y-1);
    fire (x+1,y+1);
    fire (x+2,y-1);
  end
```

The parameters of the function are the coordinates of the leftmost cell.

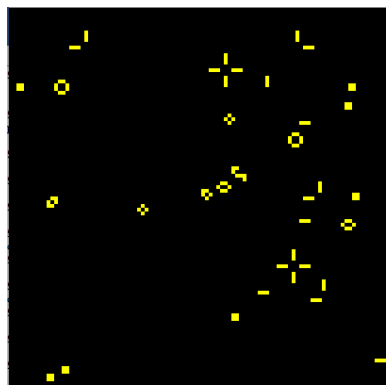
7.1.7 Main Module

The main module launches a graphical thread, creates the CA space, and places a pentomino in the middle of it:

```
let module main () =
  begin
    thread graphics (maxx,maxy,BLACK);
    create_ca ();
    r_pentomino (maxx/2,maxy/2)
  end
```

Note that the threads executing the cell behaviors are not immediately running (they actually start to run at the second instant); thus, the pentomino is correctly placed in the CA space *before* the cell threads start.

The following figure shows the result of running the program with 10000 cells (`maxx` and `maxy` both equal to 100):



7.2 Self-Replicating Loops

We consider self-replicating loops (SR loops) in cellular automata spaces, issued from the work of Langton [48]. SR loops are most of the time defined by state-transition rules described by look-up tables, possibly extended with several additional operations. This is for example the case of Sayama's *evoloop* [58, 59] whose definition has the following shape:

1. The core of the definition is a set of 258 rules of the form $(c, t, r, b, l) \rightarrow n$, where the vector (c, t, r, b, l) stores the states of the von Neumann neighborhood (c stands for the cell itself, t for top, r for right, b for bottom, and l for left) and n is the new state.

2. The previous rules are extended with their rotationally symmetric ones.

3. The rules are transformed by adding a new state, and by changing some of them in which the new state appears.

This definition is not convenient for adding further modifications, as noted by Sayama who advocates a high-level language for describing the rules. Moreover, a sheath is most often used to facilitate loop definitions; for example, in

the loop of Langton which is shown on Figure 7.1, the sheath is made of the cells numbered by 2 (in yellow). Sheath cells are however rather artificial and constitute a overhead for the simulation. The question is thus raised on how to define loops without the help of a sheath.

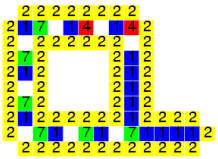


Figure 7.1: Langton's Loop

One proposes an algorithmic way for defining the cell behaviors and apply this to a variant of the evoloop. This variant is simpler (it does not have a sheath) than the initial evoloop, but still manifests an "evolutionary-like" behaviour.

7.2.1 Replication Process

Basically, SR loops are made of sequences of genes that are duplicated and interpreted during the replication process. Replication is informally described (in absence of a sheath) on Figure 7.2; pictures from 1 to 9 show the main steps of the process:

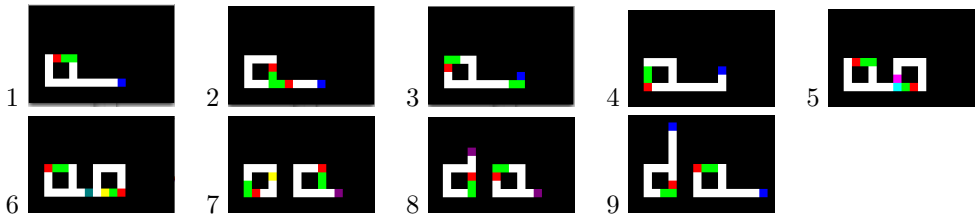


Figure 7.2: Main Steps of Loop Replication

1. This is the initial loop. It is made of a square of white, red and green cells, and of an arm starting from the right/bottom corner of the square and terminated by a blue cell. The red cell is interpreted as a gene which makes the arm turn one step in the counter-clockwise direction. The two green cells are interpreted as genes which make the arm grow one step. The genes are moving in the counter-clockwise direction along the square and the arm. The blue cell is a creator which interprets genes.

2. The genes are duplicated at the beginning of the arm (instant 10).

3. The creator (in blue) has just interpreted the turning gene (red) reaching the end of the arm (instant 15).

4. The arm has grown as the two grow genes have been interpreted (instant 17) .

5. The arm has turned back on itself. A collide cell (in magenta) is created at the end of the arm. A barrier (in cyan) is created; it replaces a green gene (instant 49).

6. A sprout cell (in yellow) is created in the new loop (instant 50). In the initial cell, the previous barrier produces a stop cell (in grey) which starts arm retraction.

7. In the new loop, a new arm has been created when the sprout has reached the right/bottom corner; the arm is now ended by a waiter cell (in pink) (instant 54). In the initial loop, the arm has finished to retract and a sprout has been created.

8. In the initial loop, a waiter cell is created by the sprout (instant 58).

9. After receiving turn genes, waiters become translators in both loops. The replication process is now completed: two loops are living (instant 69).

In [58, 59], Sayama introduces a specific destruction mechanism in order to manifest evolution-like global behaviors. Here, the destruction mechanism is nothing more than the one used for arm retraction during self replication. As in Sayama's work, an evolution-like process can appear using this destruction mechanism (see section 7.2.4).

The SR Loops and the code to simulate them are available on the Web. A first version of the SR Loops implementation is described in [20], which mainly considers the use of reactive programming to implement cellular automata.

7.2.2 Cell Implementation

One describes the code concerned with the cells. The specific behavior of SR loops is described in the next section.

Cell State

As previously, there is a distinction between *active* and *quiescent* cells. A quiescent cell should actually stay quiescent as long as its neighborhood is only composed of quiescent cells (empty neighborhood). Here, there are 3 kinds of cells: quiescent cells; cells to be erased at the next instant; *directed cells*. A directed cell basically has a direction associated to it[2]. One first defines the following two types (the type `nature_t` is defined in Section 7.2.3):

[2]Cells have four neighbors identified by their direction: top, right, bottom, left (von Neumann neighborhood).

```
type direction_t = TOP | BOTTOM | RIGHT | LEFT

type state_t = QUIESCENT | ERASE | State of direction_t * nature_t
```

Cell Module

The implementation of cells has the following features:

- Each cell is implemented as a thread, with an associated activation event.
- The implementation avoids any processing of quiescent cells, which are just waiting to be activated.
- At each instant, a cell which is not quiescent generates the activation event of each of its neighbors, with an associated information describing its own state.
- Additional information (firing orders; see below) can be transmitted to a neighbor by generating its activation event.
- After having activated its neighbours, a non-quiescent cell collects the information from them, and finally updates its state according to the received information.

Note that only active cells and cells belonging to the neighborhood of an active cell are activated. Of course, for a quiescent cell the waiting action does not imply any "busy-waiting" as long as its neighborhood is empty. The following module implements cells:

```
let module cell (x,y,activation,init,adjacent) =
  let state = local ref init in
  let fire_count = local ref 0 in
  let was_quiescent = local ref false in
    loop begin
      if is_quiescent (!state) then
        await activation
      else
        awake (adjacent,!state);
      let neighbours = local ref [4] Absent in
      begin
          fire_count := 0;
          was_quiescent := is_quiescent (!state);
          for_all_values activation
             with info -> process_info (neighbours,info,fire_count);
          if !fire_count > 0 then
            if (not !was_quiescent) || (!fire_count > 1) then
              state := ERASE
            else
              test_firing (neighbours,state)
          else
```

```
            begin
              behavior (neighbours,state,adjacent);
              generate draw_event with Image (x,y,size,state2color (!state));
            end
        end
    end
```

The local variable `state` stores the cell state and the local variable `fire_count` is used to count how many neighbors have fired the cell. The event `activation` is the cell activation event. The cell basically executes cyclically the following actions:

- If the cell is quiescent, then the activation event is awaited (primitive `await`, which is implemented in a totally passive way).
- If the cell is active, then its state is communicated to all the neighbors (function `awake`).
- All the values received through the event `activation` are then processed (function `process_info`, which assigns to `fire_count` the number of neighbors that have fired the cell).
- If the cell is not fired, then the transition function `behavior` (defined in 7.2.3) is called and the event `draw_event` is generated to draw the cell on the screen.
- If the cell is fired, there are two cases (see 7.2.2):
  - If the cell was not quiescent or if the number of firing orders is greater than one, then the cell is erased.
  - Otherwise, the cell state is changed according to the (unique) firing order (function `test_firing`).

Firing of cells

In some situations, there is the need to *fire* a neighbor, that is to transmit it an order to change its state. The actual change of state is however left to the responsibility of the fired cell, to avoid nondeterminism. Indeed, a cell which is not quiescent or which is fired several times during the same instant will be erased. Thus, a firing order is executed only if the fired cell is quiescent and if the order is unique, which entails determinism. The function `fire` is recursively defined by:

```
let fire (dir,state,adjacent) =
  match adjacent with
     Nil_list -> () |
     Cons_list (head,tail) ->
        match head with Dir_event (d,evt) ->
          if d = dir then
```

```
            generate evt with Fire (opposite (dir),state)
         else
            fire (dir,state,tail)
      end
   end
```

In this definition, `Nil_list` is the empty list, `Cons_list` is the list constructor, and `match` is the matching statement to deconstruct values. Note that termination of the function `fire` is checked by the system.

7.2.3 SR Loops Behavior

The nature of SR loop cells is defined by the following type:

```
type nature_t =
    BASIC | COLLIDE | BARRIER | GROW_GENE | TURN_GENE
  | CREATOR | SPROUT | STOP | WAITER | PRE_WAITER
```

The constants of `nature_t` are interpreted as follows:

- `BASIC`: the cell has no specific behavior.
- `COLLIDE`: the arm gets back to itself.
- `BARRIER`: to produce a sprout and to stop arm construction.
- `GROW_GENE` and `TURN_GENE`: the genes.
- `CREATOR`: the cell which interprets genes.
- `SPROUT`: starts a new arm when reaching a corner.
- `STOP`: for arm retraction, until the initial loop is reached. Stopped cells turn to quiescent at the next instant.
- `PRE_WAITER` and `WAITER`: to deal with sprouts.

A cell is said to be *dead* if its state is quiescent or `ERASE`.

The cell behavior is defined by the function `behavior` using three auxiliary functions: `creator`, `waiter`, and `step`. For these functions, one adopts the following terminology: let c be a cell whose direction is *dir*. The neighbor of c in the direction *dir* is called the *target* and the opposite cell of the target is called the *source*. The neighbor in the clockwise direction is called *clk* and the opposite cell of clk is called *invclk*.

Creator Function

The function `creator` basically translates genes. The cell concerned by the gene (the target for `GROW_GENE`, and the invclk cell for `TURN_GENE`) is first tested for death. If the cell is not dead, the current state is changed to `COLLIDE`. Otherwise, the state is changed to `BASIC` and a new creator is fired in the direction defined by the gene.

```
let creator (neighbors,dir,state,adjacent) =
   if not is_dead (neighbors,dir) then
     state := State (dir,COLLIDE)
   else if test_nature (neighbors,opposite (dir),GROW_GENE) then
     begin
       state := State (dir,BASIC);
       fire (dir,State (dir,CREATOR),adjacent)
     end
   else if test_nature (neighbors,opposite (dir),TURN_GENE) then
     begin
      state := State (dir,BASIC);
      fire (invclk (dir),State (invclk (dir),CREATOR),adjacent)
     end
   else ()
```

Waiter Function

The `waiter` function fires a waiter at the next instant, if the current cell is a pre-waiter. If the current cell is a waiter, it waits for a turn gene and fires the target cell as a creator.

```
let waiter (neighbors,dir,state,adjacent) =
   if nature_of_state (!state) = PRE_WAITER then
     begin
       state := State (dir,BASIC);
       fire (dir,State (dir,WAITER),adjacent)
     end
   else if nature_of_state (!state) = WAITER &&
           test_nature (neighbors,opposite (dir),TURN_GENE) then
     begin
       state := State (dir,BASIC);
       fire (dir,State (dir,CREATOR),adjacent)
     end
   else ()
```

Step Function

The function `step` is called for directed cells which are neither creator, nor waiter, nor pre-waiter. The following actions are performed in turn:

1. The cell becomes quiescent if it is erased.

2. The cell is erased if one of its neighbors is (erase propagation).
3. The cell becomes quiescent if it is stopped.
4. In front of a stopped cell, the cell is stopped if the invclk cell is dead. This continues arm retraction.
5. In front of a barrier with a different direction, the cell turns to `STOP`. This starts arm retraction.
6. A pre-waiter is fired when the sprout reaches a corner in which the target cell is dead.
7. The cell becomes a sprout when the source cell is a barrier. This is the case in the copy of the loop, when the arm is cut.
8. When the invclk cell is a collide cell (produced when a creator cannot progress forward), then the current cell becomes a barrier in the same direction as the collide cell. This is the situation encountered when the arm loops back on itself. The change of direction is a way to close the new loop construction.
9. The cell becomes a sprout when the target is stopped while the invclk is not dead. This ends arm retraction.
10. If none of the previous rules apply, then the state of the source cell is simply copied into the current cell.

```
let step (neighbors,state,adjacent) =
   let dir = direction_of_state (!state) in
   let source = opposite (dir) in
   let clk = clk (dir) in
   let invclk = invclk (dir) in
   if is_erase (!state) then state:= QUIESCENT
   else if neigbor_erased (neighbors,source) || neigbor_erased (neighbors,dir) ||
           neigbor_erased (neighbors,clk) || neigbor_erased (neighbors,invclk) then
      state := ERASE
   else if nature_of_state (!state) = STOP then
      state := QUIESCENT
   else if test_nature (neighbors,dir,STOP) && is_dead (neighbors,invclk) then
      state := State (dir,STOP)
   else if test_nature (neighbors,dir,BARRIER) &&
      direction_of_state (state_of_info (!neighbors[dir2int(dir)])) <> dir then
      state := State (dir,STOP)
   else if test_nature (neighbors,source,SPROUT) && is_dead (neighbors,dir) then
      fire (dir,State (dir,PRE_WAITER),adjacent)
   else if test_nature (neighbors,opposite (dir),BARRIER) then
      state := State (dir,SPROUT)
   else if test_nature (neighbors,invclk,COLLIDE) then
      state := State (clk,BARRIER)
   else if test_nature (neighbors,dir,STOP) && not (is_dead (neighbors,invclk)) then
```

```
      state := State (dir,SPROUT)
   else
     state := State (dir,
        nature_of_state (state_of_info (!neighbors[dir2int (source)])))
```

Cell Behavior

The function `behavior` changes the state of a cell accordingly to its neighbors. It performs the following actions:

1. Nothing is done if the cell is quiescent.
2. If the cell is erased or if its source is dead, the cell becomes quiescent.
3. Otherwise, the function `waiter`, `creator`, or `step` is called, according to the cell state.

```
let behavior (neighbors,state,adjacent) =
  let dir = direction_of_state (!state) in
    begin
      if is_quiescent (!state) then ()
      else if is_erase (!state) then state := QUIESCENT
      else if is_dead (neighbors,dir) then state := QUIESCENT
      else if nature_of_state (!state) = WAITER ||
              nature_of_state (!state) = PRE_WAITER then
        waiter (neighbors,dir,state,adjacent)
      else if nature_of_state (!state) = CREATOR then
        creator (neighbors,dir,state,adjacent)
      else
        step (neighbors,state,adjacent);
    end
```

Several remarks can be made:

- When a gene cannot be interpreted by the function `creator` because the target is not dead, then the state is changed to collide. At the next instant, a barrier is created (rule 8 of `step`) which will lead to the creation of a stopped cell (rule 5). Propagation of this stopped cell explains why a loop can kill another one, just by touching it.
- The creation of a new loop always results from the path:

  `SPROUT`→`PRE_WAITER`→`WAITER`→`CREATOR`.

  There are only two paths for sprout creation: either `COLLIDE`→`SPROUT` or `COLLIDE`→`STOP`→`SPROUT`. New loops thus always find their origin in collisions.
- Erased cells are dead cells. Thus, an erased cell can be fired, which may end the propagation of `ERASE`. This phenomenon explains the presence of incomplete structures.

7.2.4 Experiment

Let us consider the evo-loop of Figure 7.3 with two sequences of consecutive genes. The loop is placed in the middle of a finite[3] CA made of 100x100 cells.

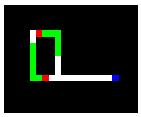


Figure 7.3: Evo Loop

On the left of Figure 7.4 is the initial situation; on the right is the situation after 1000 instants. Loops with various shapes have appeared. On the left of Figure 7.5 is the situation after 2000 instants, and on the right after 4000 instants. In the last situation, all loops with the shape of the initial evo-loop have been destroyed and the only loops that remain living are small ones with only one sequence of 3 genes. This is the "evolution-like" characteristics pointed out by Sayama: small loops are more active in the replication process and thus get a bonus over larger loops.

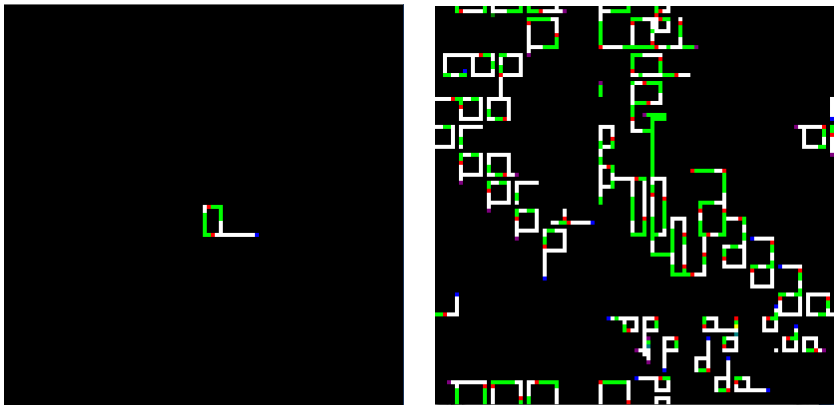


Figure 7.4: Evo Loop. Left: initial, right: 1K instants

We have also experimented the use of two synchronised schedulers on a dual-core machine. Each scheduler being executed by a dedicated native thread, the two schedulers can be run in real parallelism. However, the benefit of using a multi-core machine does not clearly appear in this case, as the processing load that each cell has to perform at each instant seems to be not heavy enough compared with the load of scheduler synchronisation. Note that this is not the case for simulations of colliding particles which are considered in [22]. It remains

[3]Periodic boundary conditions in both directions.

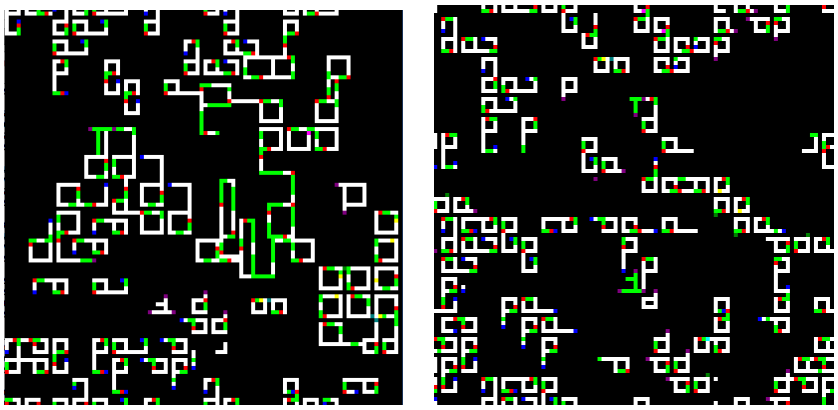


Figure 7.5: Evo Loop. Left: 2K instants, right: 4K instants

an open question to find a CA space that could really benefit from a multi-core machine.

Chapter 8

Colliding Particles

We propose to consider the graphical simulation of a set of colliding particles as a benchmark for multicore machines. Collision processing is basically an algorithm whose complexity is square in the number of particles (actually $n^2/2$, where n is the number of particles). Thus, the amount needed of computing ressource can grow very rapidly as the number of particles increases. Each particle can be involved in several concurrent steps of collision processing: there is thus a need for protecting the data associated to particles. Moreover, the parallel threads should periodically synchronise, in order to get a realistic simulation (otherwise, a subset of particles could stay idle, while another subset is animated several times).

8.1 One Scheduler

This section contains the FunLoft code for the simulation in which all particles are run by the same scheduler.

It should be noticed that threads in FunLoft (produced as instances of modules using the `thread` construct) are basically user-threads, defined at a logical level. On the opposite, FunLoft schedulers are mapped on physical (native) threads (pthreads in the current implementation). Thus, schedulers are the units executed in real parallelism on multicore machines.

8.1.1 Drawing Processor

An auxiliary type `image_t` is defined to hold images of particles.

```
type image_t = Image of
    int // x
  * int // y
  * int // size
  * color_t
```

Particles are paint as balls, using the function `draw_ball` (not described here):

```
let draw_ball_image (i) =
   match i with Image (x,y,r,c) ->
       draw_ball (x*zoom,y*zoom,r*zoom,c)
   end
```

The module `draw_processor` process drawing orders: at each instant, the values of the event `draw_event` are collected and the function `draw_image` is called for each of them:

```
let module draw_processor (draw_event) =
   loop
      for_all_values draw_event with i -> draw_ball_image (i)
```

8.1.2 Particles

A particle is a structure holding four references (holding floats): x and y coordinates, x and y speeds, and two constants: radius and color:

```
type particle_t = Particle of
      float ref      * // x coord
      float ref      * // y coord
      float ref      * // x speed
      float ref      * // y speed
      int            * // radius
      color_t          // color
```

The function `new_particle` creates a new particle which is randomly placed in the simulation, and has a random speed.

```
let max_speed = 5

let random_speed (m) =
  let x = random_int (2) in
  let v = random_int (m) + 1 in
     if x = 0 then v else -v

let new_particle (maxx,maxy,size,color) =
   let x = int2float (random_int (maxx)) in
   let y = int2float (random_int (maxy)) in
   let sx = int2float (random_speed (max_speed)) in
   let sy = int2float (random_speed (max_speed)) in
      Particle (ref x,ref y,ref sx,ref sy,size,color)
```

Two functions are defined to invert the speed of particles:

```
let invert_x_speed (s) =
   match s with Particle (_,_,sx,_,_,_) -> sx:=-.!sx end

let invert_y_speed (s) =
   match s with Particle (_,_,_,sy,_,_) -> sy:=-.!sy end
```

8.1.3 Inertia and Bouncing Behaviors

The `inertia` function gives inertia to particles by simply incrementing the coordinates by the speed:

```
let inertia (me) =
  match me with Particle (x,y,sx,sy,_,_) ->
    begin x:=!x+.!sx; y:=!y+.!sy end
  end
```

The module `bounce_behavior` makes particles bounce on the applet borders.

```
let module bounce_behavior (me,maxx,maxy) =
  match me with Particle (x,y,sx,sy,radius,_) ->
    let r = int2float (radius) in
    let d = int2float (2*radius) in
    let mx = int2float (maxx)-.r in
    let my = int2float (maxy)-.r in
    let mx2 = 2.*.mx in
    let my2 = 2.*.my in
    loop begin
      if !x <. r then
        begin invert_x_speed (me); x:=d-.!x end
      else if !x >. mx then
        begin invert_x_speed (me); x:=mx2-.!x end
      else ();
      if !y <. r then
        begin invert_y_speed (me); y:=d-.!y end
      else if !y >. my then
        begin invert_y_speed (me); y:=my2-.!y end
      else ();
      cooperate
    end
  end
```

8.1.4 Collision Behavior

First, an auxiliary type `coord_t` is defined to hold information on particles:

```
type coord_t = Coord of float * float * float * float * int
```

```
let particle2coord (p) =
   match p with Particle (x,y,sx,sy,r,_) -> Coord (!x,!y,!sx,!sy,r) end

let dot_product (d1,d2,d3,d4) = (d1*.d3) +. (d2*.d4)
```

The `collide` function process a possible collision between a particle and another one represented by its coordinates. The distance between the two particles is computed, then the collision is processed if needed.

```
let collide (me,other) =
  match me with Particle (rx1,ry1,rsx1,rsy1,rad1,_) ->
    let x1 = !rx1 in
    let y1 = !ry1 in
    let sx1 = !rsx1 in
    let sy1 = !rsy1 in
      match other with Coord (x2,y2,sx2,sy2,rad2) ->
        let max_dist = int2float (rad1+rad2) in
        let dx = x2 -. x1 in
        let dy = y2 -. y1 in
        let dist = square_root_f ((dx*.dx) +. (dy*.dy)) in
          if dist <. (max_dist /. 2.) || dist >. max_dist then
            return
          else
            let d3 = dot_product (sx1,sy1,dx,dy) in
            let d5 = d3 /. dist in
            let d6 = dot_product (sx2,sy2,-.dx,-.dy) in
            let d7 = d6 /. dist in
            let d8 = d5 +. d7 in
               if (d8 <. 0.0) || (d8 = 0.0) then return
               else
                 let dsx = d8 *. (dx /. dist) in
                 let dsy = d8 *. (dy /. dist) in
                   begin
                     rx1 := x1 -. dsx;
                     ry1 := y1 -. dsy;
                     rsx1 := sx1 -. dsx;
                     rsy1 := sy1 -. dsy;
                   end
      end
  end
```

The `process_all_collisions` maps the `collide` functions on a list of particles:

```
let process_all_collisions (me,list) =
  match list with
      Nil_list -> ()
```

```
    | Cons_list (other,tail) ->
        begin
          collide (me,other);
          process_all_collisions (me,tail);
        end
  end
```

The collision behavior can now be defined, which cyclically performs the following actions: first, a collision event is generated with its own coordinates as value; second, all the values generated for the collision event are collected and stored in a list; third, the list is processed by `process_all_collisions`; fourth, the inertia function is called. The code of the module `collide_behavior` is:

```
let module collide_behavior (me,collide_event) =
  let r = ref Nil_list in
    loop
      begin
        generate collide_event with particle2coord (me);
        get_all_values collide_event in r;
        process_all_collisions (me,!r);
        inertia (me);
      end
```

8.1.5 Draw Behavior

The draw behavior of a particle consists in generating at each instant a drawing order for the particle:

```
let module draw_behavior (me,draw_event) =
 loop
   begin
      match me with Particle (x,y,_,_,r,c) ->
         generate draw_event with Image (float2int (!x),float2int (!y),r,c)
      end;
      cooperate
   end
```

8.1.6 Particle Behavior

Each particle is animated by three threads: one for bouncing on the applet borders, one for collision processing, and one for the graphics. All these threads share the particle.

```
let module particle_behavior (collide_event,draw_event,color) =
  let s = new_particle (mx,my,size,color) in
    begin
      thread bounce_behavior (s,mx,my);
```

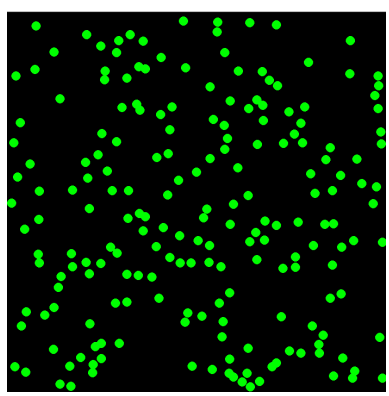
```
      thread collide_behavior (s,collide_event);
      thread draw_behavior (s,draw_event);
    end
```

8.1.7 Global System

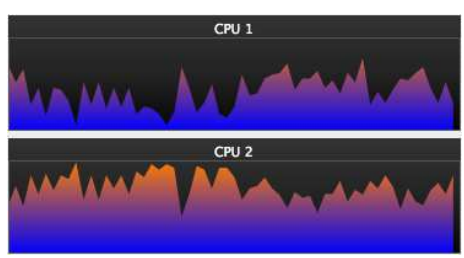
The main module first declares the two events `draw_event` and `collide_event`; then, two threads are launched, one to process the graphics, and one to collect the drawing orders; finally, the threads to animate the particles are launched:

```
let module main () =
   let draw_event = event in
   let collide_event = event in
    begin
      thread graphics (mx,my,BLACK);
      thread draw_processor (draw_event);
      repeat particle_number do
          thread particle_behavior (collide_event,draw_event,GREEN);
    end
```

The simulation obtained with 200 particles is:



The CPU usage on a dual-core machine is shown on the following screenshot:



8.2 Two Schedulers

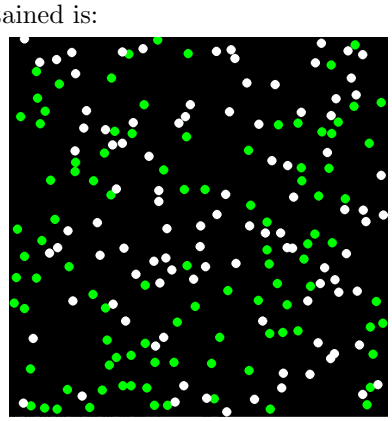
One now separate the particles in two parts, each one corresponding to a distinct scheduler. The schedulers are synchronised. As schedulers are run by dedicated native threads, they can be run in real parallelism, on a multicore machine. Note that the collision and drawing events are shared by the two schedulers. The declaration of the two synchronised scheduler is:

```
let s1 = scheduler and s2 = scheduler
```

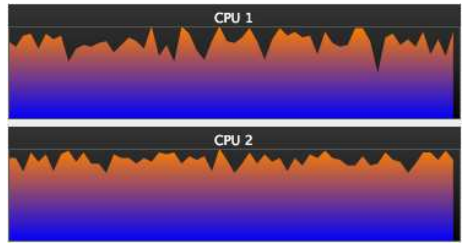
In the main module, half of the particles are launched by the main thread while linked to `s1`, and the other half while it is linked to `s2` (to distinguish the particles, they are painted in different colors). The thread for the graphics and the drawing processor are both launched in `s1`; they could as well be launched in `s2` (the point is that an error would be found if they were launched in the implicit scheduler).

```
let module main () =
  let draw_evt = event in
  let collide_evt = event in
    begin
      link s1 do
        begin
          thread graphics (mx,my,BLACK);
          thread draw_processor (draw_evt);
          repeat particle_number / 2 do
             thread particle_behavior (collide_evt,draw_evt,WHITE);
         end;
      link s2 do
        begin
         repeat particle_number / 2 do
           thread particle_behavior (collide_evt,draw_evt,GREEN);
        end
    end
```

The simulation obtained is:



The CPU usage on a dual-core machine is shown on the following screenshot:



Compared to the CPU usage in the case of a unique scheduler, one sees that the execution is really parallel. Actually, the global CPU usage is 150%.

One would get an error if the two schedulers were declared as being *not* synchronised:

```
let s1 = scheduler
let s2 = scheduler
```

The error produced indicates that a possible communication could occur through the two events:

```
type error while analysing module main:
possible communication between schedulers s1 and s2
through draw_event and collide_event
```

The machine characteristics are: MacBook Pro running Mac OS X 10.4.10, processor Intel Core 2 Duo, 2.33 GHz, 2GB of memory. Graphics is based on SDL[9]. The time for simulating 500 particles during 100 instants (one instant corresponds to the execution of all the particles) is:

500 particles	1 sched	2 scheds
real	0m21.832s	0m14.189s
user	0m21.102s	0m21.369s
sys	0m0.220s	0m0.379s

This is the output of the unix command `time`), with a memory footprint of about 70MB. Note that the total number of performed interactions is about $100 * 500^2 = 25^6$. For 1000 particles, one obtains:

1K particles	1 sched	2 scheds
real	1m20.564s	0m53.724s
user	1m19.587s	1m23.508s
sys	0m0.491s	0m1.078s

Chapter 9

Preys and Predators

The code of Chapter 8 describing a simulation of colliding particles is reused for a simulation of preys and predators. In addition to show the re-use of code, the prey/predator simulation exhibits dynamic creation of threads, a characteristics which is absent in the colliding particles simulation. Indeed, in the prey/predator simulation, when all the preys have been killed, a new generation of preys is automatically created, leading to a cyclic simulation. Note that, despite the unbounded number of threads created during execution (it increases at each generation), the number of *simultaneously living* threads is bounded.

As in the simulation of colliding particles, one considers two synchronised schedulers. Each scheduler animates half of the preys and half of the predators:

```
let sched1 = scheduler
and sched2 = scheduler
```

Several variables are defined to set the simulation parameters:

```
let pred_number = 10
let prey_number = 1000
let max_pred_speed = 4.
let max_prey_speed = 5.
let kill_dist = 10.0
let pred_visibility = 400.0
let prey_visibility = 100.0
let very_far = 10000.0
```

9.1 Sprites

Preys and predators, called *sprites*, are of the type `sprite_t`. An element of this type contains a particle, defined in Chapter 8, a boolean indicating if the sprite is living or dead, and a unit event to kill the sprite (actually, only preys can be killed). New sprites are created with random position and speed by the function `new_sprite`.

```
type sprite_t =
  Sprite of
      particle_t   * // associated particle
      bool ref     * // living flag
      unit event_t   // kill event

let new_sprite (color) =
  let p = new_particle (maxx,maxy,size,color) in
    Sprite (p,ref true,event)
```

Several accessors functions are defined to access the fields of the sprite and of its associated particle (for simplicity, one gives only the ones for the x-axis):

```
let x_coord (s) =
   match s with Sprite (p,_,_) ->
     match p with Particle (x,_,_,_,_,_) -> !x end
   end

let x_speed (s) =
   match s with Sprite (p,_,_) ->
      match p with Particle (_,_,sx,_,_,_) -> sx end
   end

let living_sprite (s) =
   match s with Sprite (_,l,_) -> !l end
```

9.2 Sprite Descriptors

A sprite descriptor contains the position and speed of the associated particle, together with the kill event of the sprite. The descriptor of a sprite is created with the function `sprite2descriptor`.

```
type descriptor_t =
  Descriptor of
      float      *    // x coord
      float      *    // y coord
      float      *    // x speed
      float      *    // y speed
      unit event_t    // kill event

let sprite2descriptor (s) =
   match s with Sprite (p,_,k) ->
      match p with Particle (x,y,sx,sy,_,_) ->
          Descriptor (!x,!y,!sx,!sy,k)
      end
   end
```

A void descriptor is defined with zero coordinates, and the function `reachable` tests if a descriptor is void.

```
let void_descriptor = Descriptor (0.0,0.0,0.0,0.0,event)

let reachable (d) =
  match d with Descriptor (x,y,_,_,_) ->
     x <> 0.0 && y <> 0.0
  end
```

9.3 Simple Behaviours

The drawing behavior generates the event `draw_event` with the image of the sprite (re-using the function `draw` of particles), while the sprite is alive:

```
let module sprite_draw_behavior (me,draw_event) =
  while living_sprite (me) do
  begin
    match me with Sprite (p,_,_) -> draw (p,draw_event) end;
    cooperate
  end
```

The inertia behavior gives inertia to living sprites, re-using the function `inertia` of particles:

```
let module sprite_inertia_behavior (me) =
   while living_sprite (me) do
    begin
      match me with Sprite (p,_,_) -> inertia (p) end;
      cooperate
    end
```

The bouncing and colliding behaviours are defined in the same way; we do not describe here for the sake of simplicity.

9.4 Chasing Behaviour

Several auxiliary functions are first defined. The function `distance` returns the distance between a sprite and a point defined by its coordinates:

```
let distance (s,x,y) =
  let dx = x -. x_coord (s) in
  let dy = y -. y_coord (s) in
     square_root_f ((dx*.dx)+.(dy*.dy))
```

The function `add_speed` increments the speed of a sprite by a difference of coordinates; moreover, the new speed obtained is limited to a maximal value:

```
let add_speed (dist,me,oth,speed,max) =
  let d = oth -. me in
    begin
      speed := !speed +. (d /. dist);
      if !speed >. max then speed := max end;
      if !speed <. -. max then speed := -.max end;
    end
```

The function `detect` finds the closest target, present in the visibility area of a sprite:

```
let detect (me,descriptor,visibility,min,target) =
  match descriptor with Descriptor (x,y,sx,sy,_) ->
    let d = distance (me,x,y) in
      if (d <. visibility) && (d <. !min) then
         begin target := descriptor; min := d end
      else ()
  end
```

The `chase` function evaluates the distance to a prey and kills the prey, if possible, by generating its kill event. Otherwise, the predator changes its speed to move in direction of the prey, using te function `add_speed` previously defined:

```
let chase (me,descriptor) =
  match descriptor with Descriptor (x,y,_,_,k) ->
   let dist = distance (me,x,y) in
    if dist <. kill_dist then
       generate k
    else
      begin
         add_speed (dist,x_coord (me),x,x_speed (me),max_pred_speed);
         add_speed (dist,y_coord (me),y,y_speed (me),max_pred_speed);
       end
  end
```

Then, the chasing behavior can be defined using the previous auxiliary functions. At each instant, the predator generates the `pred_event` with a descriptor of it, and from the values generated by the preys with the event `prey_event`, it determines which the closest one; finally, if a prey has been detected, the predator chases it. The code is:

```
let module chase_behavior (me,pred_evt,prey_evt) =
  let min = ref 0.0 in
  let target = ref void_descriptor in
    loop
      begin
        generate pred_evt with sprite2descriptor (me);
```

```
        min := very_far;
        target := void_descriptor;
        for_all_values prey_evt
           with prey -> detect (me,prey,pred_visibility,min,target);
        if reachable (!target) then chase (me,!target) end
      end
```

The run-away behaviour is defined in a similar way, and is not described here, for simplicity.

9.5 Death of Preys

One now defines the death behaviour which awaits the killing event of a sprite and then unsets its living flag; in this way, all associated behaviours will be forced to terminate.

```
let module sprite_death_behavior (me) =
  match me with Sprite (_,l,k) ->
    begin
      await k;
      l:=false;
    end
  end
```

Each sprite is animated by three threads, one for bouncing on the simulation borders, one for processing the killing event, and one for drawing the sprite:

```
let common_function (collide_evt,color,draw_event) =
  let s = new_sprite (color) in
    begin
      thread sprite_bounce_behavior (s);
      thread sprite_death_behavior (s);
      thread sprite_draw_behavior (s,draw_event);
      s
    end
```

Moreover, predators are processing collisions and are chasing, while preys are only running away (they do not collide):

```
let predator_function (collide_evt,pred_evt,prey_evt,draw_event) =
  let s = common_function (collide_evt,RED,draw_event) in
    begin
      thread sprite_collide_behavior (s,collide_evt);
      thread chase_behavior (s,pred_evt,prey_evt);
    end
```

```
let prey_function (collide_evt,pred_evt,prey_evt,color,draw_event) =
  let s = common_function (collide_evt,color,draw_event) in
    begin
      thread sprite_inertia_behavior (s);
      thread runaway_behavior (s,pred_evt,prey_evt);
    end
```

9.6 Automatic Creation of Preys

One now turn to the creation of preys. The `launch` function is first defined, which launches half of the preys:

```
let launch (collide_evt,pred_evt,prey_evt,color,draw_event) =
   repeat prey_number / 2 do
      prey_function (collide_evt,pred_evt,prey_evt,color,draw_event)
```

To launch the totality of preys, the module `automatic_preys` links to each scheduler and calls the `launch` function. A `join` instruction is introduced, which blocks the control until all previously launched preys have been killed. The code is:

```
let module automatic_preys (collide_evt,pred_evt,prey_evt,draw_event) =
    loop
      join
        begin
          link sched1 do
            launch (collide_evt,pred_evt,prey_evt,CYAN,draw_event);
          link sched2 do
            launch (collide_evt,pred_evt,prey_evt,GREEN,draw_event);
        end
```

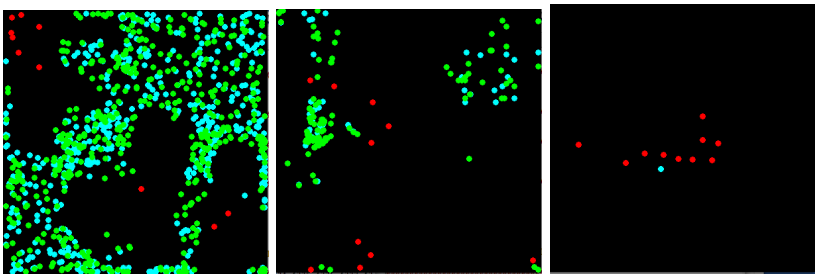
Note that the creation of threads is authorised in the loop body because it is under the control of `join`. All the preys launched are actually killed when the loop body terminates, which make the loop safe.

9.7 Main Module

The main module starts by defining four events, one generated by the preys to signal their position, one for the predators, one for processing predator collisions, and one for drawing sprites. Then, two instances of the `graphics` and `draw_processor` modules are launched in the scheduler `sched1` (the choice is arbitrary, as `sched1` and `sched2` are synchronised). Half of the predators are then launched in each of the schedulers, and finally, an instance of the module `automatic_preys` is launched (actually, in the implicit scheduler).

```
let module main () =
  let pred_evt = event in
  let prey_evt = event in
  let collide_evt = event in
  let draw_event = event in
     begin
       link sched1 do
         begin
           thread graphics (maxx,maxy,BLACK);
           thread draw_processor (draw_event);
         end;
       link sched1 do
         repeat pred_number / 2 do
           predator_function (collide_evt,pred_evt,prey_evt,draw_event);
       link sched2 do
         repeat pred_number / 2 do
           predator_function (collide_evt,pred_evt,prey_evt,draw_event);
       thread automatic_preys (collide_evt,pred_evt,prey_evt,draw_event);
     end
```

Here is a sequence of snapshots of the simulation of size 400x400 (1000 preys, 10 predators):



Note that, by replacing in the behavior of preys the inertia behavior by a collision behavior, one gets a variant in which preys cannot overlap but which needs more computing resources.

Chapter 10

Data Flow Programming

In this chapter, one considers a very specific programming model, called *dataflow*, and the question of its implementation in FunLoft. At the basis of the dataflow model are the notions of *channel* and of *process*. A channel is basically a list of data, used to connect processes, managed in a "first-in/first-out" way. Processes are made of sequential code, and are run in parallel. Systems are basically graphs of processes interconnected by channels. Systems can be recursively defined (thus possibly leading to infinite graphs).

In the dataflow model, there are several ways to exhaust the memory: data used by processes can be of unbounded sizes; channels can store unbounded numbers of data; recursively defined systems can be infinite. This basically means that *the data flow model cannot be implemented in FunLoft*. Indeed, only data whose sizes can be proved to be bounded are definable in FunLoft.

In order to implement the dataflow model, one thus should *relax* the controls made by the FunLoft compiler, and basically accept the possibility of memory leaks. Actually, one implements the dataflow model by relaxing two controls: first, recursively defined modules become allowed; second, the stratification-based controls which entail bounds on the size of data are turned-off. Note, however that the controls on the memory accesses, and the controls insuring that instants indeed terminate (reactivity) are still maintained. Note also that the basic atomicity of sequential instructions is preserved.

One considers now the question: are the channels the only means of communication between processes? There are actually two responses. In the *weak* dataflow model, processes are allowed to communicate through shared memory, while this is forbidden in the *strong* model. Thus, the response is "no" in the weak model and "yes" in the strong one. Here, only the weak model is considered, and the issue of implementing the strong model in FunLoft is left open.

First, the channels are described in 10.1. Then, some processes are given in 10.2, which are used in the following sections. Section 10.3 is an example of a finite system which computes the fibonnacci numbers. Section 10.4 describes a sieve to produce prime numbers. A sieve to produce lucky numbers is described

in 10.5. The case of numbers which are both prime and lucky is considered in 10.6.

10.1 Channels

In order to simplify, one supposes that channels only contain integer values. A channel is implemented as a structure made of an event (the triggering event), a reference on an integer (the channel length), and a reference on a list (the channel content):

```
type ‘a channel_t =
    Channel of unit event_t * int ref * ‘a list ref
```

Three auxiliary functions are defined to manipulate channels: `new_channel` returns a new empty channel; `length` returns the length of a channel; `get_signal` returns the triggering event:

```
let new_channel () =
  Channel (event,ref 0,ref Nil_list)

let length (chan) =
  match chan with Channel (_,len,_) -> !len end

let get_signal (chan) =
  match chan with Channel (sig,_,_) -> sig end
```

10.1.1 Put

Channels are basically unbounded FIFO files. One defines the `push` function to add a value in a channel. As channels are unbounded, this action is always immediately possible (this is why `push` can be implemented as a function). The `push` function calls the recursively defined function `append` which places a value at the end of a list; this function is proved to always terminate.

```
let append (v,l) =
  match l with
     Nil_list -> Cons_list (v,Nil_list) |
     Cons_list (a,b) -> Cons_list (a,append (v,b))
  end

let push (chan,v) =
  match chan with Channel (_,len,list) ->
    begin
      len++;
      list := append (v,!list);
    end
  end
```

The `put` function pushes a value in a channel and generates the triggering event of the channel to possibly awake the channel consumer:

```
let put (channel,v) =
  begin
    push (channel,v);
    generate get_signal (channel)
  end
```

10.1.2 Get

The function `extract` returns the first value available from a channel. Actually, this is the head of the list of values. The value 0 is arbitrary returned from an empty channel; note however that extraction from an empty channel should never occur.

```
let extract (chan) =
   match chan with Channel (_,len,l) ->
     let list = !l in
       match list with Cons_list (head,tail) ->
         begin
            len--;
            l := tail;
            head
         end
      | default -> 0
   end
```

The instruction executed to get a value from a channel is non-atomic as it can lasts several instants; it is thus not possible to implement it as a function. We choose not to use a module, but rather to use a macro-definition to define the instruction. This is possible in FunLoft, because a first pre-processing phase is always performed by the FunLoft compiler (cf Chapter 11) before parsing code files. The macro-definition of `get` is the following:

```
#define get(channel,res) \
 begin \
   while length (channel) = 0 do \
   begin \
      await get_signal (channel); \
      cooperate \
   end; \
   res := extract (channel) \
 end
```

10.2 Processes

One now defines several processes, used in the sequel. Of course, many other processes can be coded in a similar way. The first process is the process `const` that produces at each instant the same value on its output channel:

```
let module const (val,output) =
  loop
    begin
      put (output,val);
      cooperate
    end
```

The process `producer` produces on its output channel a new value at each instant:

```
let module producer (from,incr,output) =
 let v = ref from in
   loop
     begin
       put (output,!v);
       v := !v+incr;
       cooperate
     end
```

The process `dup` duplicates its input on its two outputs:

```
let module dup (input,output1,output2) =
 let v = ref 0 in
   loop
     begin
       get (input,v);
       put (output1,!v);
       put (output2,!v);
       cooperate
     end
```

The process `print` prints the values of its input, one at each instant (the values are separated with a blank character and the standard output is flushed):

```
let module print (input) =
 let v = ref 0 in
   loop
     begin
       get (input,v);
       print_int (!v); print_string (" "); flush ();
       cooperate
     end
```

The module `follow` outputs the first value of its left input, then it copies its right input on the output:

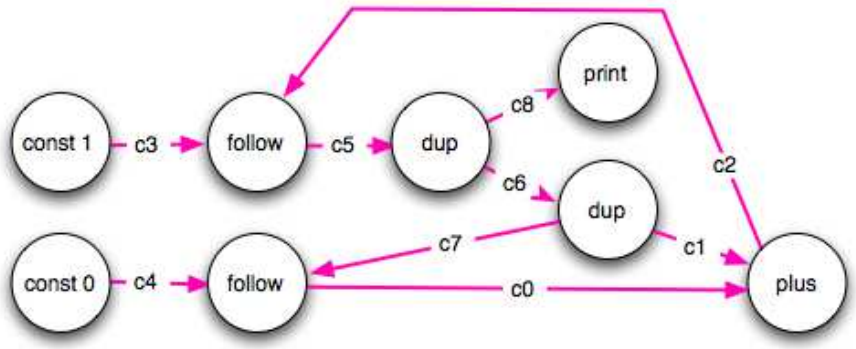
```
let module follow (left,right,out) =
  let v = ref 0 in
    begin
      get (left,v);
      put (out,!v);
      loop
        begin
          get (right,v);
          put (out,!v);
          cooperate
        end
    end
```

The module `plus` gets the values of its two inputs and outputs the sum on the output:

```
let module plus (left,right,out) =
  let v1 = ref 0 in
  let v2 = ref 0 in
    loop
      begin
        get (left,v1);
        get (right,v2);
        put (out,!v1 + !v2);
        cooperate
      end
```

10.3 Fibonnacci Numbers

One considers a program which prints the list of fibonnacci numbers, defined by: $F_0 = 1$, $F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$. The program is rather surprising as it only involves the processes `plus2`, `const`, `dup`, and `fol` previously defined. The program is:




```
let module main () =
  let c0 = new_channel () in
  let c1 = new_channel () in
  let c2 = new_channel () in
  let c3 = new_channel () in
  let c4 = new_channel () in
  let c5 = new_channel () in
  let c6 = new_channel () in
  let c7 = new_channel () in
  let c8 = new_channel () in
    begin
      thread plus   (c0,c1,c2);
      thread const  (1,c3);
      thread const  (0,c4);
      thread follow (c3,c2,c5);
      thread dup    (c5,c8,c6);
      thread dup    (c6,c7,c1);
      thread follow (c4,c7,c0);
      thread print  (c8)
    end
```

10.4 Prime Numbers

In this section, one considers *sieves* which are algorithms for producing numbers with a given characteristics. The most well-known sieve is of course the sieve of Eratosthenes for producing prime numbers. Actually, there are two types of sieves: the number of elements of bounded sieves is statically fixed, while in unbounded sieves the number of elements is potentially infinite. Only unbounded sieves are considered here.

10.4.1 Standard Eratosthenes Sieve

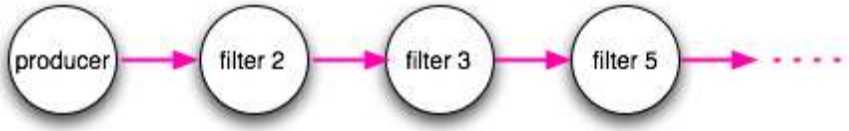
The basic version of the (unbounded) sieve of Eratosthenes can be implemented by the following C function[1]:

```
int main ()
{
   list_t primes = init_list ();
   int current = 3, create = 1;
   while (1) {
      cell_t cell = primes->first;
      while (cell != NULL) {
         if (multiple_of (current,cell->val)) {
            create = 0; break;
         }
         cell = cell->next;
      }
      if (create) {
         add_list (primes,current);
         output (current);
      } else create = 1;
      current += 2;
   }
   return 0;
}
```

The type `list_t` is the type of lists made of elements of type `cell_t` which contains an integer field `val` and a pointer `next` to the next cell. The definition of these two types is standard in C and not given here.

10.4.2 Eratosthenes Sieve in FunLoft

The sieve is recusively defined. Actually, a producer of numbers is linked to a sequence of filters; each filter rejects the multiple of a fixed number. The system looks like:



[1] thanks to Fabrice Peix

Filter

Multiples of a number are filtered by the following `filter` module which only outputs the values of its input that are not multiple of a given number. Multiples are checked using the `remainder_int` function:

```
let remainder_int (num,denum) =
  let d = num / denum in
    num - (d*denum)

let module filter (num,input,out) =
  let v = ref 0 in
    loop
      begin
        get (input,v);
        if not remainder_int (!v,num) = 0 then
          put (out,!v)
        end;
        cooperate
      end
```

Sift Process

The `sift` process is recursively defined. It basically gets the first value v of its input and prints it; then, it creates a new filter connected to the input for filtering the multiples of v and having as output a new channel *intern*; finally, it recursively creates a new `sift` process whose input is *intern*. The definition is:

```
let module sift (input,pr) =
  let v = ref 0 in
      begin
        get (input,v);
        put (pr,!v);
        let intern = new_channel () in
          begin
            thread filter (!v,input,intern);
            thread sift (intern,pr);
          end
      end
```

Sieve

The producer of prime numbers consists in a producer (starting from 2, with 1 as increment), connected with a `sift` process. The output channel is given as a parameter to the module:

```
let module primes (out) =
  let intern = new_channel () in
    begin
      thread producer (2,1,intern);
      thread sift (intern,out);
    end
```

The main module launches a producer of prime numbers connected to a process to print them:

```
let module main  () =
  let out = new_channel () in
    begin
      thread primes (out);
      thread print (out)
    end
```

The program prints the list of prime numbers, creating a new filter for each prime. The beginning of the output is:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163
167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 ...
```

Of course in such a algorithm, the memory will eventually exhaust at some time; this is basically why this algorithm is rejected by the standard FunLoft compiler (and thus, why it must be compiled using specific options).

10.5 Lucky Numbers

Lucky numbers [36] are generated by a sieve which is very close to the one of Eratosthenes[2]. The production of lucky numbers is defined by the following algorithm: write all the odd numbers: 1,3,5,7,9,11,... The first number greater than 1 is 3, so *delete* every third number: 1,3,7,9,13,15,19,... The first number greater than 3 in the list is 7, so *delete* every seventh number: 1,3,7,9,13,15,21,25,31,... And so on, without ending. Numbers that remain are called *lucky* numbers.

The sieve to produce lucky numbers is based on a filtering function which filters numbers according to their indexes, and not to their values as in the Eratosthenes sieve.

```
let module filter_lucky (i,num,input,out) =
  let v = ref 0 in
  let n = ref i in
    loop
      begin
```

[2]See [7] for all that concerns integer sequences.

```
        get (input,v);
        if not remainder_int (!n,num) = 0 then
            put (out,!v)
        end;
        n++;
        cooperate
      end
```

The sift function is defined by:

```
let module sift_lucky (i,input,pr) =
  let v = ref 0 in
      begin
        get (input,v);
        put (pr,!v);
        let intern = new_channel () in
          begin
            thread filter_lucky (i+1,!v,input,intern);
            thread sift_lucky (i+1,intern,pr);
          end
      end
```

The sieve to produce the lucky numbers is defined by:

```
let module luckies (out) =
  let intern = new_channel () in
    begin
      put (out,1);
      thread producer (3,2,intern);
      thread sift_lucky (2,intern,out);
    end
```

Note the first 1 which is directly put on the output. The producer starts from 3, with 2 as increment; this is a slight optimisation to avoid the processing of even numbers. The first index of `sift_lucky` is 2 because the first value, 3, is actually the second one (after the first 1). The list of lucky numbers starts with:

```
1 3 7 9 13 15 21 25 31 33 37 43 49 51 63 67 69 73 75 79 87 93
99 105 111 115 127 129 133 135 141 151 159 163 169 171 189 193
195 201 205 211 219 223 231 235 237 241 259 261 267 273 283 ...
```

10.6 Lucky Prime Numbers

Let us now consider the numbers that are simultaneously prime and lucky. The direct production of these lucky prime numbers is not so clear using a unique sieve (what would be the filtering function?). However, a possible solution is to

reuse the two previous sieves and to build a system made of two sub-systems, one producing prime numbers and the other producing lucky numbers. In such a solution, a third part is needed to compare the produced numbers and to detect those produced by both sieves. This is the role of the module `cmp` defined by:

```
let module cmp (l,p,out) =
  let hold = ref -1 in
  let v = ref 0 in
  let input = ref l in
    loop
      begin
        get (!input,v);
        if !hold = !v then
          put (out,!hold)
        else if !hold < !v then
          begin
            hold := !v;
            if !input = l then input := p else input := l end
          end;
        cooperate;
      end
```

The system is the following:

```
let module main () =
  let l = new_channel () in
  let p = new_channel () in
  let out = new_channel () in
  begin
    thread luckies (l);
    thread primes (p);
    thread cmp (l,p,out);
    thread print (out);
  end
```

The list of lucky prime numbers produced is:

```
3 7 13 31 37 43 67 73 79 127 151 163 193 211 223 241 283 307
331 349 367 409 421 433 463 487 541 577 601 613 619 631 643
673 727 739 769 787 823 883 937 991 997 1009 1021 1039 ...
```

10.6.1 Two Schedulers

One now consider the use of two schedulers, one for computing primes numbers and one for computing lucky numbers. The two schedulers can be run in parallel, which increases efficiency.

```
let p_sched = scheduler
let l_sched = scheduler
```

The previous comparison thread cannot be used as it, because its two input channels are not situated on the same scheduler. One must thus makes the comparison thread migrate from one scheduler to the other to check for numbers which are produced in both schedulers. In order to share the code, one defines a macro with a scheduler in parameter (in the present version, `link` instructions can only concern already existing schedulers, not parameters):

```
#define EXHAUST(sched) \
  let v = local ref 0 in \
  let new = local ref false in \
  let hold = !h in \
    begin \
       link sched do \
          let term = local ref false in \
             while not !term do \
              begin \
                 get (input,v); \
                 if hold = !v then \
                    begin new:=true; term:=true end \
                 else if hold < !v then \
                    term:=true \
                 end; \
                 cooperate; \
              end; \
       h:=!v; \
       if !new then put (output,!v) end; \
   end
```

The reference `h` contains the value `hold` which is checked for existence in `sched`. First, the thread links to the `sched` and cyclically gets values from the input channel, until a value greater or equal to `hold` is found. Then, the thread ends to get values from the input channel and re-links to the previous scheduler. If a value equal to `hold` is found, then it is output as being produced by both sieves. Otherwise, `h` is changed by the value found.

Two modules are defined for exhausting the channels that collect the produced values:

```
let module exhaust_p (h,input,output) = EXHAUST (p_sched)
let module exhaust_l (h,input,output) = EXHAUST (l_sched)
```

The comparison module is the following:

```
let module cmp (l,p,out) =
  let hold = ref -1 in
```

```
  loop
    begin
      run exhaust_p (hold,p,out);
      run exhaust_l (hold,l,out);
    end
```

Note that `hold` and `output` are both accessed by `exhaust_p` and `exhaust_l`; however the compiler verifies that no access to these references is made while linked to `p_sched` or `l_sched`.

The main program is:

```
let module main () =
  let l = new_channel () in
  let p = new_channel () in
  let out = new_channel () in
    begin
      link l_sched do thread luckies (l);
      link p_sched do thread primes (p);
      thread cmp (l,p,out);
      thread print (out);
    end
```

Here are the measures (using the `time` Unix command) for producing[3] the 500 first lucky-prime numbers (in approximatively 500MB of memory; the last produced number is 29989):

500 first LP	1 sched	2 scheds
real	1m16.684s	0m21.464s
user	1m15.227s	0m36.818s
sys	0m1.069s	0m1.300s

Note the full usage of the two CPUs when two schedulers are defined. This is actually an ideal situation where the load of CPU needed for synchronisation (the moves of the `cmp` thread) is very small compared to the load of CPU needed for computing each sieve.

10.7 Related Work

The work considered in this chapter is strongly related to dataflow-based synchronous languages. Actually, most of these languages only consider the static case; for example, in Lustre [30] recursive definitions of systems are forbidden. Moreover, a static analysis, called *clock calculus*, insures that channels always have bounded sizes (actually, the bound is 1, for all channels). The Lucid Synchrone language [55] allows recursive definitions of systems, while maintaining

[3]Machine characteristics: MacBook Pro running Mac OS X 10.4.10, processor Intel Core 2 Duo, 2.33 GHz, 2GB of memory.

the existence of bounds for channel sizes. Thus, the existence of static checks to insure that channels have bounded sizes is certainly a strength of synchronous languages, compared to FunLoft.

However, the termination of instants is not proved in these languages. Moreover, only the weak model is implemented, and preservation of atomicity is not checked. Finally, implementations of synchronous languages are presently not able to benefit from multicore machines.

10.8 Conclusion

The implementation described in this chapter is very straightforward: actually, it is only based on pre-processing. However, controls on the size of the memory used (and on the number of simultaneously living threads) must be disconnected. However, the basic atomicity of sequential execution and the reactivity of programs are still valid. Moreover, programs can benefit from multicore machines, as shown in the last example, where lucky-prime numbers are computed by two sieves run in parallel on a dual-core machine.

Two points should be of interest: first, to implement the strong dataflow model, instead of the weak one. This would need to go deeper in the implementation code, and maybe imply to design specific type systems for dealing with channels and processes. Second, it could be interesting to implement checks to prove that channels are of bounded size. The work on Lucid Synchrone (for the possibility to define infinite networks of processes) would certainly be of great interest for that purpose.

Chapter 11

The Compiler

11.1 The Fl Compiler

In order to be processed by the compiler, FunLoft code should be placed in files ended with `.fl`. The compiling command is named `fl`, and the simplest form of compilation is:

```
fl tst.fl
```

This command compiles in C the file `tst.fl`, and then calls the C compiler (presently, the Gnu `gcc` version 4.0.1 compiler), producing an executable file named `a.out`. As in C, the name of the executable can be changed using the `-o` option.

C code in `.c` files and compiled code in `.o` files can also be given to the FunLoft compiler, which transmits them directly to the back-end C compiler.

There are two formats for the comments: either the standard C forms `/* ... */` and `// ...`, or the Ocaml form `(* ... *)`. In both cases, comments can be nested.

As the executable file `a.out` is produced silently, the compiler complains with a warning message when `a.out` cannot be produced because no `main` module is defined.

11.1.1 Pre-processing

A first pre-processing pass is performed on the `.fl` files. The pre-processor used in the current version of the compiler is the one of the C compiler `gcc`[1]. One can thus, in `.fl` files, use macro-processor directives and macros exactly as in C. For example, one can include files using a `#include` directives, as in

```
#include "trace_inst.fl"
#include "external.fl"
```

[1] `Actually: gcc -P -E`

```
#include "graphics.fl"
#include "drawing_processor.fl"
....
```

The inclusion of `.fl` files gives a (very rough) way to reuse FunLoft code.

Pre-processing directives of the form `-D` and `-I` can be given to the FunLoft compiler; they are transmitted to the preprocessor and also to the back-end C compiler. For example, let us consider the following re-definition of the printing of integers:

```
#ifdef VERBOSE
#define PR(m) \
   begin \
     print_string ("the result is: ");\
     print_int (m);\
     print_string ("\n");\
   end
#else
#define PR(m) print_int (m)
#endif
```

The following compilation command switches to the verbose mode, while compiling `tst.fl`:

```
fl -DVERBOSE tst.fl
```

11.1.2 C Functions

Interfacing with C code is possible using extern functions of FunLoft. For example, here is an exemple of FunLoft program using an extern function called `c_print`:

```
let c_print : string -> unit

let module main () =
  begin
    c_print ("ok\n");
    quit (0);
  end
```

All the basic types of FunLoft are implemented as one unique C type, called `value` and declared in a file named `val.h`. For example, here is a C definition of the function `c_print`:

```
#include <stdio.h>
#include "val.h"
```

```
value c_print (value s)
{
    printf ("%s",val2string (s));
    fflush (stdout);
    return val_unit;
}
```

The predefined function `val2string`, declared in the file `val.h`, converts a `value` data into a `char*` value of C.

Extern functions should be correctly defined in C; they should neither crash nor block, and they should always terminate. This is the programmer responsability to insure this. A warning message is produced to remind this.

11.1.3 Extern Constants

Extern constants should be set in C by the function `external_constants` which is by default defined as a function with an empty body. For example, let's consider the following program that prints an extern string:

```
let c_msg : string

let module main () =
  begin
    print_string (c_msg);
    quit (0);
  end
```

The extern string `c_msg` should be set in `extern_constant`, called at the very beginning of each execution:

```
#include <stdio.h>
#include "val.h"

value c_msg;

void extern_constants (void)
{
    c_msg = string2val ("ok\n");
}
```

Extern constants should be correctly set in C as data of the type `value`, otherwise the program could crash. This is the programmer responsability to insure this.

11.1.4 Arguments of Main

The arguments given to `a.out` are placed in an array of strings, whose first element is the command name. Thus, the following program prints its name, followed by the list of the command parameters:

```
let module main (argv) =
  begin
    let c = ref 0 in
      repeat dimension (argv) do
        begin
          print_string ("<");
          print_string (!argv[!c]);
          print_string (">");
          c++;
        end;
    print_string ("\n");
    quit (0);
  end
```

Execution of the command "`a.out a 1 b`" produces the output `<a.out><a><1><b>`.

11.1.5 Options

In addition to the previous options, the following ones are available:

`-help`	print this message and terminate
`-version`	print version of fl and terminate
`-debug`	print information on what should be done
`-v`	print information on what is done
`-full_type`	full information for types
`-no_main_defined`	no warning if main module not defined
`-no_stratification`	stratification switched off
`-allow_recursive_module`	allow recursive modules
`-allow_all_recursive_functions`	allow all recursive functions
`-allow_thread_in_loop`	allow uncontrolled thread creation

The fourth last options deserve a special status because they are switching-off controls to check that memory remains bounded. In case they are used, memory leaks become possible. These options should thus be used only when bounded memory is not a mandatory property; this is for example the case with the dataflow implementation of Chapter 10; here are the makefile targets to compile the Fibonnacci numbers and the prime numbers (the value of the `FL` macro is the FunLoft compiler):

```
fib:
    $(FL) -no_stratification fib.fl

primes:
    $(FL) -no_stratification -allow_recursive_module prime.fl
```

11.2 Structure of the Compiler

The compiler[2] is organised in four stages:

- Parsing of FunLoft code is performed using the standard tools lex and yacc.
- The processor `fl2loft` translates the FunLoft code in Loft. Loft is a language which implements the FairThreads model in C.
- The processor `loft` compiles the Loft code issued from the previous stage. It produces C code using the FairThreads library in C.
- Finally the standard C compiler compiles the C code produced, links it to the FairThreads library and to the standard Pthreads library, and finally produces an executable.

The `fl` compiler is thus based on two associated softwares: the Loft compiler `loft`, and the FairThreads library `fthread`.

To show how the compiler works, let us use the `-debug` option:

```
fl tst.fl -debug
```

The result is:

```
(cp tst.fl _tmp_.c
gcc -C -P -E  _tmp_.c > _tmp_..c
cat lib/predef.fl _tmp_..c > _tmp_.fl
bin/fl2loft  < _tmp_.fl > tst.fl.loft)
&&
bin/loft -Iinclude -Llib  tst.fl.loft
     -lfl -lloft_ft -lfthread -lpthread -lgc -lm
```

First, the `tst.fl` file is copied in a `.c` file, in order to be processed by the C preprocessor; this produces the temporary file `_temp_..c`. Second, some definitions (for example, the definition of the type `list`) are placed at the beginning of the previous file, using the Unix `cat` command. Third, Loft code is produced, using the `fl2loft` command. When there is no error, the Loft compiler is called. Note that several libraries are used: `fl`, `loft_ft`, and `fthread` for using FairThreads; `pthread` for using Posix threads; `gc` for using the GC, and `m` for mathematical functions. The GC is the one of Hans-J. Boehm [3].

11.3 Typing Information

The processor `fl2loft` produces typing information in the produced Loft code. This information is output as comments. It may help to the programmer to access it. For example, consider the following list of definitions:

[2] The current version is v0.2.1.

```
let x = ref 0
let incr (x) = x++
let module m () = incr (x)
```

Here is the corresponding typing information, produced before the Loft code:

```
/************* analysis of var x *************/
/* TYPE ANALYSIS
x: int ref
*/
/************* analysis of function incr *************/
/* TYPE ANALYSIS
incr: int ref -> unit
*/
/************* analysis of module m *************/
/* TYPE ANALYSIS
m_create:  unit -> thread_t
*/
```

When the `-full_type` is set, the information is more complete. Information about effects is also printed. For example, from the same example, the typing information becomes:

```
************* analysis of var x *************/
/* TYPE ANALYSIS
x: int ref(‘k117,public,‘x118)
*/
/************* analysis of function incr *************/
/* TYPE ANALYSIS
incr: int ref(‘k123,‘s122,‘x124) -> unit
(written:int ref(‘k123,‘s122,‘x124)
 read:int ref(‘k123,‘s122,‘x124)
 accessed:int ref(‘k123,‘s122,‘x124)
 events:) []
*/
/************* analysis of module m *************/
/* TYPE ANALYSIS
m_create:  unit -> thread_t
(written:int ref(‘k117,public,‘x118)
 read:int ref(‘k117,public,‘x118)
 accessed:int ref(‘k117,public,‘x118)
 events:) []
*/
```

The information added to the type of `x` is:

- the stratification level of the reference is `k117`;

- the reference is public;
- the scheduler area to which the reference belongs is `x118`.

Associated to functions and modules are *effects*, which are decomposed in a list of written references, a list of read references, a list of accessed references (reference which are both read and written), and a list of used events. There is also a list of constraints (between []) which we will not consider here.

11.4 Error Messages

Error messages are a big issue for languages in which types are infered, which is the case of FunLoft. Actually, an error is detected when a contradiction is detected between the definition of an object and a use of it. However, intermediate information can be involved in the unification process that leads to the error detection. The problem is thus, in case of error, to give the initial definition of the objet and the real place where it is used faultly.

Chapter 12

Related Work

One considers three domains related to FunLoft. The first is the one of threads which either appear as libraries or are incorporated in existing languages. The second domain is the synchronous-reactive approach. The third domain contains approaches in which computations are decomposed into small pieces that can be combined to build complex behaviors.

12.1 Thread Based Formalisms

12.1.1 Thread Libraries in C

Several thread libraries exist for C. Among them, the PTH library [53] implements the POSIX standard for preemptive threads. LinuxThreads [5] is an implementation of PTH for Linux, based on native (kernel-level) threads.

Quick Threads [47] provides programmers with a minimal support for multi-threading at user-space level. Basically, it implements context-switching in assembly code, and is thus a low-level solution to multi-threading.

Gnu Portable Threads [34] (GNU Pth) is a library of purely cooperative threads which has portability as main objective. The Next Generation POSIX Threading project [6] proposes to extend GNU Pth to the M:N model (M user threads, N native threads), with Linux SMP machines as target.

12.1.2 Java Threads

Java introduce threads at language level. Actually, threads are generally heavily used in Java, for example when graphics or networking is involved. No assumption is made on the way threads are scheduled (cooperative or preemptive schedulings are both possible) which makes Java multi-threaded systems difficult to program and to port [44]. This difficulty is pointed out by the suppression from the version 1.2 of the language of the primitives to gain fine control over threads [4]. These primitives actually corresponds to the `stop`, `suspend`, and `resume`.

The use of Java threads with multiprocessor architectures leads to several difficulties which are actually under study in the Java community. One of these difficulties is called the *Double Check Locking* (DCL) which concerns the Java memory model (JMM) and is rather tricky (see [] for details). These studies should lead to some changes of the Java thread model in the next versions (1.5) of Java.

It should be mentioned that the initial version of FairThreads has been proposed in the context of the Java language [19] in order to simplify concurrent programming in Java; this version was however limited to cooperative threads.

12.1.3 Threads in Functional Languages

Threads are used in several ML-based languages such as CML [57]. CML is preemptively scheduled and threads communication is synchronous and based on channels. Threads are also introduced in CAML [2]; they are implemented by time-sharing on a single processor, and thus cannot benefit from multiprocessor machines.

FairThreads has been recently introduced in the Bigloo [1] implementation of Scheme. The present version does not support unlinked threads, but special constructs are introduced to deal with non-blocking cooperative I/Os.

12.2 Reactive Approach

FunLoft actually belong to the so-called *reactive approach* [8] which is issued from *synchronous languages*. One first compare synchronous languages and the reactive approach before describing several reactive programming languages.

12.2.1 Synchronous Languages vs. Reactive Programming

To the family of synchronous languages [38] belong several programming languages which all share the same notion of an instant which is supposed to be of zero duration. In this context, the output of a program at a given instant is synchronous with its input; the synchronous characterization of these languages comes from this hypothesis.

Lustre and Signal are two data-flow synchronous languages in which one programs nets of operators, in a style very close to the one of section 10. At the basis of these dataflow languages are the nets of processes of [46].

Following the StateCharts of D. Harel, several systems have been proposed for synchronous graphical programming, among which are SyncCharts and Argos.

The Esterel synchronous language [] introduces broadcast events, called *signals*, in an imperative style. However, in Esterel, the absence of an event can be decided immediately and consequences of this absence can take place in the very same instant. This leads to "causality problems" as for example in `present S`

`else emit S end`. In this program, indeed, the signal `S` is emitted if it is absent during the instant, which is of course contradictory.

As opposite to Esterel, in reactive programming the absence of an event during one instant cannot be decided before the end of this very instant. As a consequence, the reaction to the absence of one event is delayed to the next instant. This is a way to solve the causality problems which are obstacles to modularity.

All synchronous languages are *static* in the sense that they disallow the creation at run time of new parallel components and of new events. Moreover, they do not give users any way to deal with multiprocessor machines. However, synchronous languages generally put the focus on hardware circuits in which dynamic creation does not appear. This is a major difference with reactive programming which limits to software systems.

A major claim of synchronous languages is that they make possible proofs and validations of programs. This is a consequence of the existence of a formal semantics and of the static characteristics of these languages. Program proofs and validations have not yet be considered in reactive programming.

12.2.2 Reactive-C

The Reactive-C [18] language was the first proposal for reactive programming in C. Reactive-C proposes instants and the `merge` operator which implements deterministic parallelism. A reaction of the instruction `merge i1 i2` consists in a reaction of `i1` followed in the same instant by a reaction of `i2`. Thus, one has a deterministic left-right operator with which parallelism can be introduced at any level. Reactive-C does not define events, as they are defined in FunLoft. Reactive-C is basically implemented as a straightforward preprocessor of C.

Reactive-C introduces primitives for remote execution of reactive programs over the network. Remote execution is based on the use of the *Remote Procedure Call* (RPC) mechanism. Distribution is not yet possible in FunLoft.

12.2.3 Reactive Programming in Java

The reactive approach has been implemented in Java in several ways. The first one is the SugarCubes [25] framework which is a set of classes for reactive programming in Java. Several related formalisms have been designed, among which are Junior [39] and Rejo [11]. The implementation of FunLoft is strongly linked to the implementation of the Java Junior framework [40].

Rejo is a language which is proposed for high-level reactive programming in a Java based language. Rejo is implemented on top of Junior and thus can benefit from the various implementations of it. As opposite to SugarCubes and Junior, standard Java code can be freely mixed with reactive code. Rejo also introduces primitives for migration of reactive code, implemented using the serialization and the RMI mechanisms of Java.

12.3 Approaches based on Decomposition

12.3.1 Chores and Filaments

Chores [33] and *filaments* [50] are small pieces of code that do not have private stack and are never preempted. Chores and filaments are designed for fine-grain parallelism programming on shared-memory machines. Chores and filaments are completely executed and cannot be suspended nor resumed. Generally, a pool of threads is devoted to execute them. Chores and chunk-based techniques are described in details in the context of the Java language in [31] and [44]. Automata in FunLoft are close to chores and filaments, but give programmers more freedom for direct coding of states-based algorithms. Automata are also related to *mode automata* [52] in which states capture the notion of a running mode in the context of the synchronous language Lustre [38].

12.3.2 Cohorts and Staged Computation

Cohort scheduling [49] dynamically reorganizes series of computations on items in an input stream, so that similar computations on different items execute consecutively. Staged computation intends to replace threads. In the staged model, a program is constructed from a collection of stages and each stage has scheduling autonomy to control the order in which operations are executed. Stages are thus very close to instants of FunLoft and cohort scheduling looks very much like cooperative scheduling. In the staged model, emphasis is put on the way to exploit program locality by grouping similar operations in cohorts that are executed at the same stage; in this way, cohorts and staged computations fall in the family of dataflow models.

12.4 Transactions

The transactional approach (transactional memory) gives an alternative to the use of locks in preemptive programming. There are however difficulties to use transactions when concurrent or parallel entities are highly communicating. This approach thus appears to be only a partial solution to the issues raised by parallelism. An interesting question would be to compare in detail the transactional approach with the one of FunLoft. The starting point for this work would be the AME model of [10]. An idea would be to consider the execution of FunLoft threads in parallel (rather than sequentializing their execution) using transactions, in the context of a unique scheduler. This approach seems to be a good candidate to study the relation with AME. Another interesting question we would like to investigate is: how would transactions interfere with the synchronous notion of instant?

12.5 OS

Some design choices of the Singularity [35] operating system present similarities with the concurrency model of FunLoft. In particular, a logical notion of process is at the basis of the model of FunLoft, and the separation of memory among behaviors running in parallel is guaranteed by the means of static analysis tools. It would be interesting to see how, following an approach similar to the one of Singularity, the programming model of FunLoft could benefit from operating system facilities.

12.6 Sπ-calculus

The Sπ-calculus [12] can be seen as the process calculus basis of FunLoft. There are however several differences, among which: the Sπ-calculus does not consider references, but only events (signals); functions and modules are not distinguished in the Sπ-calculus; thus, functions can take instants and be recursively defined.

Chapter 13

Conclusion

FunLoft is a new programming language, with several characteristics:

- Concurrency in FunLoft is not based on the standard premptive-everywhere technique, which leads to over-complex and unclear programming. Instead, a mix of premptive and cooperative techniques is used. Premptive scheduling is pushed aside, and used only for system sub-parts that have necessarilly to be run asynchronously. Otherwise, reactive programming, based on cooperative scheduling, is the technique used. This technique has a crucial advantage: the atomicity of sequential programming, which programmers intuitively relies on, is automatic preserved. Thus, the space of possible logical interleavings of instructions belonging to different concurrent programs is drastically reduced. This makes the semantics of programs simpler and the reasonning on them much more easier. In addition to a simpler semantics, there is no more need to use low-level constructs, such as locks, for the protection of data from interferences due to unwanted interleavings.

- Strong program properties are insured by static analyses. This limits the need of run-time checks, which are consuming the CPU resource, and which raise the issue of determining what is the wanted continuation in presence of a run-time error.

  Amongst the insured properties, some are safety properties: they basically correspond to insure that cooperation is indeed effective in synchronous parts. Reactivity (the passing of instants) belong to this kind of properties. Without safety, programming becomes much more difficult and dangerous.

  A second kind of properties concern the control of the asynchrony of accesses to the memory. These properties insure the absence of memory interferences of asynchronous sub-parts (schedulers, and unlinked threads) between them, and with synchronous sub-parts. These properties means that atomicity of sequential programming in preserved not only in synchronous sub-parts, but actually *everywhere in the system*.

A third kind properties are related to resource control and basically assess that the memory always remains bounded (by a function of the size of the program inputs). The associated checks performed by the compiler can be switched-off in certain situations, when the algorithm which is implemented requires by definition an unbounded memory (the sieve of Eratosthenes of Chapter 10 is an example of this).

At syntax level, FunLoft is a functional language with imperative features, restricted to first-order functions. User-defined types are inductive types. The first-order limitation basically comes from the need, to insure reactivity, to check for the termination of functions recursively defined on parameters of inductive types. The technique used (which could however certainly be improved) does not permits to consider higher-order functions. Types are infered which simplifies programming, but certainly makes debugging more complex. The hope is that the trade-off between the two points of view is acceptable and manageable.

Concerning concurrency, FunLoft uses brodcast signals for the communication and the synchronisation of threads linked to the same scheduler. The absence of "causality errors", usually found in synchonous languages, relies on the impossibility of immediate reaction to the absence of signals; this assumption basically comes from the reactive programming approach.

Non-determinism has three sources in FunLoft: the first one is the asynchrony due to the presence of several schedulers and unlinked threads; the second one source results from the arbitrary choice left to schedulers to choose the order in which linked threads should be run; the third sources relies on the way values generated during the same instant and for the same event are combined.

Dynamicity is extremely controlled, in order to conciliate with the bounded memory property. The number of threads that are simultaneously active in the system is bounded.

The current implementation of FunLoft is experimental in several aspects:

- it allocates too much, creating a lot of intermediate values.
- it relies on a general GC with difficulties to interface with.
- it produces rather rough error messages.

Future Work

Several tracks could be followed for future work:

- In Funloft, one considers only the existence of bounds (for example, on the number of simultaneously active threads), and not the values of these bounds. This is certainly insufficient for most practical purposes. For example, a number of simultaneously running threads bounded by the larger integer is admissible while of course not realistic. Checking for polynomial bounds should certainly be an improvment (while, yet, not totally satisfactory).

- The tests for function termination are rather rough. In particular, termination is not checked through the decreasing of integer parameters. Thus, the standard factorial function is rejected in FunLoft. This could certainly be improved in several ways.

- The number of schedulers is statically fixed. This is certainly an issue as schedulers are naturally mapped to cores. The first step would be to allow schedulers to be elements of arrays, accessed by their index. A second step would be to allow the dynamic creation of schedulers; in this context the issue is, like for the threads, to control the creation of schedulers.

- Modularity is very poor in FunLoft. Sets of related definitions should be introduced (in the spirit, for example, of the Ocaml notion of module). Object orientation could also be considered. Introduction of higher-order functions is also an issue that could be worth considering. Note however that it raises the problem of checking termination of higher-order functions.

- A distribution facility should certainly be introduced in FunLoft, needing the notion of a distributed scheduler. The memory of a distributed scheduler, compared to the one of standard schedulers (either asynchronous or synchronised), is totally separated from others. Thus, the system must be able to statically check this separation. A second issue is the communication between distributed schedulers. How should this be defined? Anyway, it needs some kind of safe marshalling/unmarshalling facility.

- The `link` instruction can be seen as a kind of primitive migration. In a distributed setting (as considered in the previous item), it should be extended to a full-fledged migration facility. A first idea would be to define the notion of an *agent* which can migrate from between distributed schedulers. At migration time, an agent should not have any possibility of accessing the memory of the left scheduler. At the arrival of the remote scheduler, it should have a way to get access to the memory of this scheduler. This implies a kind of dynamic linking, which of course should be safe.

- Garbage collection is a central issue for functionnal languages. It is reckognised as a difficult problem in a distributed context. A question is to determine how the existence of instants and the memory separation induced by FunLoft could facilitate the design of a GC adapted to the language. The separation of the scheduler memories is not absolute: a reference `r1` of a scheduler `s1` can points to a reference `r2` of another scheduler `s2`; the system just checks that `r2` cannot be accessed while linked to `s1`. Thus, a first problem is that each scheduler cannot perform its GC activity independently of the others.Another issue is how non-mutable values should be collected (or in other words, where are they created)?

Bibliography

[1] Bigloo: http://www.inria.fr/mimosa/fp/Bigloo.

[2] Caml: http://caml.inria.fr/ocaml.

[3] H. Boehm GC: http://www.hpl.hp.com/personal/Hans_Boehm/gc.

[4] Java: http://java.sun.com.

[5] LinuxThreads: http://gallium.inria.fr/~xleroy/linuxthreads.

[6] Next Generation POSIX Threading: http://www.ibm.com/developerworks.

[7] On-Line Encyclopedia of Integer Sequences: http://www.research.att.com/~njas/sequences/Seis.html.

[8] Reactive Programming: http://www-sop.inria.fr/mimosa/rp.

[9] Simple Directmedia Layer: http://www.libsdl.org.

[10] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *POPL*, pages 63–74, 2008.

[11] Raul Acosta-Bermejo. Rejo - langage d'objets ractifs et d'agents. *Thse de doctorat de l'ENSMP*, October 2003.

[12] Roberto M. Amadio. A Synchronous pi-Calculus. *Journal of Information and Computation*, 205(9):1470–1490, 2007.

[13] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[14] K. Arnold and J. Goslin. *The Java Programming Language*. Addison-Wesley, 1996.

[15] R. Bernhard, G. Berry, F. Boussinot, G. Gonthier, A. Ressouche, J-P. Rigault, and J-M. Tanzi. Programming a Reflex Game in Esterel v3.

[16] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[17] A.D. Birrell. An Introduction to Programming with Threads. *Digital Report*, (35), 1989.

[18] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4):401–428, april 1991.

[19] F. Boussinot. *Java Fair Threads*. Inria research report, RR-4139, 2001.

[20] F. Boussinot. Reactive Programming of Cellular Automata. *Inria research report, RR-5183*, May 2004.

[21] F. Boussinot. FairThreads: Mixing Cooperative and Preemptive Threads in C. *Concurrency and Computation: Practice and Experience*, 18:445–469, 2005.

[22] F. Boussinot and F. Dabrowski. Safe Reactive Programming: the FunLoft Proposal. In *Proc. of MULTIPROG – First Workshop on Programmability Issues for Multi-Core Computers*. Göteborg, January 2008.

[23] F. Boussinot and R. De Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9):1270–1282, september 1991.

[24] F. Boussinot and R. De Simone. The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, april 1996.

[25] F. Boussinot and J-F. Susini. The SugarCubes tool box - a reactive Java framework. *Software Practice and Experience*, 28(14):1531–1550, december 1998.

[26] F. Boussinot and J-F. Susini. Java Threads and SugarCubes. *Software Practice and Experience*, 30(14):545–566, 2000.

[27] P. Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, 1975.

[28] P. Brinch Hansen. Java's Insecure Parallelism. *ACM Sigplan Notices*, 34(4), 1999.

[29] P. Brinch Hansen. *The Origin of Concurrent Programming*. Springer, 2002.

[30] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. In *POPL*, pages 178–188, 1987.

[31] Thomas W. Christopher and George K. Thiruvathukal. *High Performance Java Platform Computing: Multithreaded and Networked Programming*. Sun Microsystems Press Java Series, Prentice Hall, 2001.

[32] E.W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, 1971.

[33] Derek L. Eager and John Zahorjan. Chores: Enhanced run-time support for shared memory parallel computing. *ACM Transaction on Computer Systems*, 11(1), february 1993.

[34] Ralf S. Engelschall. *Portable Multithreading*. Proc. USENIX Annual Technical Conference, San Diego, California, 2000.

[35] Hunt G. and Larus J. Rethinking the Software Stack. *Operating Systems Review, ACM SIGOPS*, 41(2):37–49, April 2007.

[36] M. Gardner. Lucky numbers and 2187. *Math. Intell.*, 19(26), 1997.

[37] D. Grossman. Type-safe Multithreading in Cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[38] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, New York, 1993.

[39] L. Hazard, J-F. Susini, and F. Boussinot. The junior reactive kernel. *Inria Research Report, RR-3732*, july 1999.

[40] L. Hazard, J-F. Susini, and F. Boussinot. Programming with junior. *Inria Research Report, RR-4027*, 2000.

[41] C.A.R. Hoare. Toward a Theory of Parallel Programming. *Operating System Techniques*, 1971.

[42] C.A.R. Hoare. Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[43] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[44] A. Hollub. *Taming Java Threads*. Apress, 2000.

[45] G. Kahn and D.B. MacQueen. Coroutines and Networks of Parallel Processes. *Proceeding of the IFIP'77 Congress*, 1977.

[46] G. Kahn and D.B. MacQueen. Coroutines and Networks of Parallel Processes. *Proceeding of the IFIP'77 Congress*, 1977.

[47] D. Keppel. Tools and techniques for building fast portable threads packages. *Technical Report UWCSE 93-05-06, University of Washington*, 1993.

[48] C. G. Langton. Self-reproduction in Cellular Automata. *Physica D*, 10:135–144, 1984.

[49] James R. Larus and Parkes Michael. Using Cohort Scheduling to Enhance Server Performance. *Proc. of USENIX Conference, Monterey Cal.*, pages 103–114, 2002.

[50] David K. Lowenthal, Vincent W. Freech, and Gregory R. Andrews. Efficient support for fine-grain parallelism on shared-memory machines. *TR 96-1, University of Arizona*, january 1996.

[51] L. Mandel and M. Pouzet. ReactiveML, a Reactive Extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[52] F. Maraninchi and Y. Remond. Running-modes of real-time systems: A case-study with mode-automata. *Proc. 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden*, 2000.

[53] B. Nichols, D. Buttlar, and Proulx Farrell J. *Pthreads Programming.* O'Reilly, 1996.

[54] G. Plotkin. A Structural Approach to Operational Semantics. *Aarhus University Report DAIMI FN-19*, 1981.

[55] M. Pouzet. Lucid Synchrone, version 3. Tutorial and Reference Manual. *Université Paris-Sud, LRI*, April 2006.

[56] R. Pucella. Reactive Programming in Standard ML. *Proceedings of the IEEE International Conference on Computer Languages (ICCL'98)*, pages 48–57, 1998.

[57] J.H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[58] H. Sayama. *Constructing Evolutionary Systems on a Simple Deterministic Cellular Automata Space.* Phd, University of Tokyo, Department of Information Science, 1998.

[59] H. Sayama. Introduction of Structural Dissolution into Langton's Self-Reproducing Loop. In *C. Adami, R.K. Belew, H. Kitano, C.E. Taylor (Eds), Artificial Life, Proc. of the Sixth International Conference on Artificial Life*, pages 114–122, 1998.

[60] M. Serrano, F. Boussinot, and B. 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.

[61] J. Trevor, J.G. Morrisett, D. Grossman, M.W. Hicks, J. Cheney, and Y. Wang. Cyclone: a Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

Contents

1 Introduction **3**

2 Language Overview **13**
2.1 Overview of the Model 13
2.2 Overview of the Language 17

3 Language Description **21**
3.1 Definitions 21
3.1.1 Variables 21
3.1.2 Types 21
3.1.3 Functions 22
3.1.4 External Definitions 23
3.1.5 Schedulers 23
3.1.6 Modules 24
3.2 Values and Expressions 25
3.2.1 Basic Types 25
3.2.2 Expressions 26
3.2.3 Predefined Functions 27
3.3 Instructions 28
3.3.1 General Instructions 28
3.3.2 Infinite Loops 29
3.3.3 Cooperate 29
3.3.4 Await 30
3.3.5 Collecting Generated Values 31
3.3.6 Join and Run 32
3.3.7 Unlink 32
3.3.8 Link 33

4 Static Analyses **35**
4.1 Reactivity 35
4.2 Memory Separation 37
4.3 Resource Control 39
4.4 Control on Parametric Types 42

5 Programming Style **43**
5.1 Nondeterminism . . . 43
5.1.1 Internal Nondeterminism . . . 44
5.1.2 External Nondeterminism . . . 45
5.1.3 Communicable Nondeterminism . . . 47
5.2 Data Protection . . . 48
5.2.1 Multi-instant Processing . . . 49
5.2.2 Multiple Data . . . 50
5.2.3 Protection by Threads . . . 51
5.3 Run-time Errors . . . 52
5.3.1 Division by Zero . . . 52
5.3.2 Out of Bounds in Arrays . . . 53

6 Basic Examples **55**
6.1 Mutual Stops . . . 55
6.2 Wait/Notify . . . 56
6.2.1 Avoiding Busy-Waiting . . . 57
6.2.2 Notify All . . . 58
6.2.3 Targeted Notification . . . 59
6.3 Synchronisation Barriers . . . 61
6.4 Readers/Writers . . . 61
6.4.1 Writer . . . 62
6.4.2 Readers . . . 63
6.5 Producers/Consumers . . . 63
6.5.1 Unique area . . . 63
6.5.2 Two areas . . . 64
6.6 Reflex Game . . . 65
6.7 Sine + Cosine = Circle . . . 71

7 Cellular Automata **75**
7.1 Game of Life . . . 75
7.1.1 Cells . . . 75
7.1.2 CA Space . . . 76
7.1.3 Living Strategy . . . 77
7.1.4 Creation of CA . . . 78
7.1.5 Graphics . . . 78
7.1.6 Initial Configuration . . . 79
7.1.7 Main Module . . . 79
7.2 Self-Replicating Loops . . . 80
7.2.1 Replication Process . . . 81
7.2.2 Cell Implementation . . . 82
7.2.3 SR Loops Behavior . . . 85
7.2.4 Experiment . . . 89

8 Colliding Particles **91**
8.1 One Scheduler . . . 91
8.1.1 Drawing Processor . . . 91
8.1.2 Particles . . . 92
8.1.3 Inertia and Bouncing Behaviors . . . 93
8.1.4 Collision Behavior . . . 93
8.1.5 Draw Behavior . . . 95
8.1.6 Particle Behavior . . . 95
8.1.7 Global System . . . 96
8.2 Two Schedulers . . . 97

9 Preys and Predators **99**
9.1 Sprites . . . 99
9.2 Sprite Descriptors . . . 100
9.3 Simple Behaviours . . . 101
9.4 Chasing Behaviour . . . 101
9.5 Death of Preys . . . 103
9.6 Automatic Creation of Preys . . . 104
9.7 Main Module . . . 104

10 Data Flow Programming **107**
10.1 Channels . . . 108
10.1.1 Put . . . 108
10.1.2 Get . . . 109
10.2 Processes . . . 110
10.3 Fibonnacci Numbers . . . 111
10.4 Prime Numbers . . . 112
10.4.1 Standard Eratosthenes Sieve . . . 113
10.4.2 Eratosthenes Sieve in FunLoft . . . 113
10.5 Lucky Numbers . . . 115
10.6 Lucky Prime Numbers . . . 116
10.6.1 Two Schedulers . . . 117
10.7 Related Work . . . 119
10.8 Conclusion . . . 120

11 The Compiler **121**
11.1 The Fl Compiler . . . 121
11.1.1 Pre-processing . . . 121
11.1.2 C Functions . . . 122
11.1.3 Extern Constants . . . 123
11.1.4 Arguments of Main . . . 123
11.1.5 Options . . . 124
11.2 Structure of the Compiler . . . 125
11.3 Typing Information . . . 125
11.4 Error Messages . . . 127

12 Related Work **129**
12.1 Thread Based Formalisms 129
12.1.1 Thread Libraries in C 129
12.1.2 Java Threads 129
12.1.3 Threads in Functional Languages 130
12.2 Reactive Approach 130
12.2.1 Synchronous Languages vs. Reactive Programming 130
12.2.2 Reactive-C 131
12.2.3 Reactive Programming in Java 131
12.3 Approaches based on Decomposition 132
12.3.1 Chores and Filaments 132
12.3.2 Cohorts and Staged Computation 132
12.4 Transactions 132
12.5 OS 133
12.6 Sπ-calculus 133

13 Conclusion **135**