JoCaml: a Language for Concurrent Distributed and Mobile Programming

Cédric Fournet[1], Fabrice Le Fessant[2]
Luc Maranget[2], and Alan Schmitt[2]

[1] Microsoft Research
[2] INRIA Rocquencourt

Abstract. In these lecture notes, we give an overview of concurrent, distributed, and mobile programming using JoCaml. JoCaml is an extension of the Objective Caml language. It extends OCaml with support for concurrency and synchronization, the distributed execution of programs, and the dynamic relocation of active program fragments during execution.

The programming model of JoCaml is based on the join calculus. This model is characterized by an explicit notion of locality, a strict adherence to local synchronization, and a natural embedding of functional programming à la ML. Local synchronization means that messages always travel to a set destination, and can interact only after they reach that destination; this is required for an efficient asynchronous implementation. Specifically, the join calculus uses ML's function bindings and pattern-matching on messages to express local synchronizations.

The lectures and lab sessions illustrate how to use JoCaml to program concurrent and distributed applications in a much higher-level fashion than the traditional threads-and-locks approach.

1 An Overview of JoCaml

Wide-area distributed systems have become an important part of modern programming, yet most distributed programs are still written using traditional languages designed for closed, sequential architectures. In practice, distribution issues are typically relegated to system libraries, external scripts, and informal design patterns [4,19], with little support in the language for asynchrony and concurrency. Conversely, the distributed constructs, when present, are constrained by the local programming model, with for instance a natural bias towards RPCs or RMIs rather than asynchronous message passing, and a tendency to hide these issues behind sequential, single-threaded interfaces.

Needless to say, distributed programs are usually hard to write, much harder to understand and to relate to their specifications, and almost impossible to debug. This is due to essential difficulties, such as non-determinism, asynchrony, and node failures. Nonetheless, it should be possible to provide some language support and tools to facilitate distributed programming.

Proceedings of the Fourth Summer School on Advanced Functional Programming
St Anne's College, Oxford, 19–24 August 2002. To appear in LNCS.
© Springer-Verlag 2002

JoCaml is an attempt to provide such a high-level language in a functional setting, with linguistic support for asynchronous, distributed, and mobile programming [6,15] . JoCaml is based on the join calculus [7,8], a simple and well-defined model of concurrency similar to the pi calculus [22,21] but more suitable for programming. The join calculus is the core language for JoCaml and its predecessors, and has inspired the design of several other languages [3,23,27,28]. More formally, the join calculus is also a specification language, that can be used to state and prove the properties of programs, such as the correctness of an implementation [10,1].

JoCaml is an extension of Objective Caml 1.07 [20], a typed programming language in the ML family with a mix of functional, imperative, and object-oriented features. JoCaml extends OCaml, in the sense that OCaml programs and libraries are just a special kind of JoCaml programs and libraries. JoCaml also implements strong mobility and provides support for distributed execution, including a dynamic linker and a distributed garbage collector.

These notes give an overview of JoCaml, and how it can be used for both concurrent and distributed programming. First, we survey the basic ideas behind JoCaml as regards concurrent and distributed programming. Then, we introduce JoCaml constructs, their syntax, typing, and informal semantics, first in a concurrent but local setting (Section 2), and finally in a distributed setting (Section 3).

Starting from Objective Caml. High-level distributed programming mostly relies on scripting languages (Agent-Tcl [11], TACOMA [14], Telescript [29]). Such languages are often specialized for some specific task or architecture, and may offer poor performances for other operations, typically relegated to external calls in other languages. Besides, for the sake of flexibility, these languages don't provide much structure, such as modules, interfaces, classes, and user-defined types, to develop and maintain complex software projects.

JoCaml is based on Objective Caml (OCaml), a compiled, general purpose, high-level programming language, which combines functional, imperative and object-oriented programming styles. OCaml is a great language, with several features that are especially relevant for our purpose:

– Programs are *statically typed*, in a reasonably expressive type system with parametric polymorphism, type inference, subtyping for objects, user-defined data-types, and a rich module system.
  This is especially important in distributed systems, where there are many opportunities to assemble inconsistent pieces of software, and where debugging runtime type errors is problematic.
– As a programming environment, OCaml provides both native-code and byte-code compilers, with separate compilation and flexible linking. The latter compiler is important to implement code mobility at runtime.
– The OCaml runtime has good support for system programming, such as the ability to marshal and unmarshal any data types, even between heterogeneous platforms; an efficient garbage collector, which we have extended for

distributed collection; and low-level access to the system, with many Unix system calls (POSIX threads, sockets, ...).

Indeed, OCaml has been used to develop many complex programs for distributed systems, such as Web browsers with applets (MMM [26]), group communication libraries (Ensemble [12]), and Active Networks (SwitchWare [2]), and to experiment on a variety of parallel machines.

Adding Concurrent Programming. OCaml is a sequential language: expressions are evaluated in call-by-value, in a deterministic manner. As a first language extension, JoCaml provides support for lightweight concurrency, message passing, and message-based synchronization.

To begin with, we introduce a new expression, `spawn` *process* `;` *expression*, that executes *process* and evaluates *expression* in parallel. The respective operations in *process* and *expression* run independently; their interleaving is a first source of non-determinism. Also, *process* is not quite an expression—it is not meant to return a result—so we introduce a new syntactic class for (asynchronous) processes that is recursively defined with (synchronous) expressions.

Processes can be seen as virtual threads, running in parallel, in no particular order. The JoCaml compiler and runtime are responsible for mapping these processes to a few system threads.

Instead of producing values, processes interact by sending asynchronous messages on channels.[3] Indeed, an asynchronous message is itself a process. Accordingly, JoCaml also introduces channels and local channel definitions for processes, much like functions and `let fun` bindings for expressions:

– Channels are first-class values, with a communication type, which can be used to form expressions and send messages (either as the message address or as message content).
– Channel definitions bind channel names, with a static scope, and associate guarded processes with these names. Whenever messages are passed on these names, copies of these processes are fired.

So far, our extension still lacks expressiveness. We can generate concurrent computations but, conversely, there is no means of joining together the results of such computations or, for that matter, of having any kind of interaction between spawned processes. We need some synchronization primitives.

A whole slew of stateful primitives have been proposed for encapsulating various forms of inter-process interaction: concurrent variables, semaphores, message passing, futures, rendez-vous, monitors, ... just to name a few. JoCaml distinguishes itself by using that basic staple of ML programming, *definition by pattern-matching*, to provide a declarative means for specifying inter-process synchronization, thus leaving state inside processes, where it rightfully belongs.

[3] In addition to message passing, processes can still cause any OCaml side-effects, such as writing a shared mutable cell in a sub-expression; however, these effects are usually harder to trace than messages.

Concretely, this is done by allowing the joint definition of several channels by matching concurrent message patterns on these channels; in a nutshell, by allowing parallel composition on the left-hand-side of channel definitions.

These synchronization patterns, first introduced in the join calculus, are equivalent to more traditional forms of message passing (with dynamic senders and receivers on every channel) in terms of expressiveness, but offer several advantages from a programming language viewpoint.

1. For each channel definition, all synchronization patterns are statically known, so they can be efficiently compiled using for instance state automata [17].
2. Similarly, type systems can analyze all contravariant occurrences of channels, then generalize their types (cf. Section 2.5).
3. As regards distributed implementations, the static definition of channels (also known as *locality* in concurrency) enables an efficient implementation of routing: for a given channel, there exists a single definition that can handle the message, and the machine that hosts the definition is the only place where the message can participate to a synchronous step.

(Section 2 will illustrate the use of join patterns, and relate them to several other synchronization primitives.)

At this stage, we have a natural extension of (impure) functional programming with lightweight concurrency and synchronization. Next, we explain how this language can be used across several machines on an asynchronous network, with distributed message passing and process mobility.

Adding Distribution and Mobility. Before discussing any form of communication between JoCaml programs, we refine our model and give an explicit account of locality. In particular, we must be able to represent several runtimes and their local processes on the same network. In the join calculus, the basic unit of locality is called a "location".

Locations have a nested structure, so that a given location can recursively contain sub-locations. This model is adequate to represent a network architecture, where OS processes are running inside a computer, computers are gathered inside LANs, themselves included inside Autonomous Systems, themselves organized in a hierarchical model.

In a JoCaml executable, the whole program is itself a location, called the root location, and implicitly managed by the runtime. Additional locations can be explicitly created, within an existing location, using a special declaration that describes the content of the new location (running processes, channels, even sub-locations) and gives it a fresh location name. Hence, a distributed configuration of machines running JoCaml programs can be seen as a location tree, each location hosting its own definitions and processes.

As a first approximation, *locations are transparent*: channels have a global lexical scope, so that any process that has received a channel name can use it to send messages, and can forward it to other processes, independently of the location that defines the name. Said otherwise, from any given configuration, we could in principle erase all location boundaries and obtain a single "global location" that contains all definitions and processes, and we would still get the same communications.

In addition, *locations can be used to control locality.* Specifically, locations and location names have multiple roles in JoCaml programs:

– Locations represent units of mobility. At any time, a location and its live content can decide to migrate from its current machine to another one; this can be used to model mobile objects, threads, applets, agents . . . Conversely, location names can be passed in messages, then used as "target addresses" for such migrations. This is especially convenient to relocate parts of a program as part of the computation.
– Locations represent atomic units of failure, and can be used as targets for failure detection. It is possible to halt a location, atomically stopping the execution of all the processes and sub-locations included in the location. This can be used to discards parts of the computation without restarting the whole configuration. Besides, low-level system failures can be cleanly reflected in terms of "spontaneous" halts of root locations. Conversely, location names can also be used to detect the failure of remote locations, and trigger some failure recovery code.

Further reading. The latest JoCaml release contains a reference manual, a series of sample programs, and a more extensive tutorial [15]. Some aspects of the implementation are described elsewhere in more details: the distributed garbage collector [18,16]; the type system [9]; the compilation of join patterns [17].

A gentle introduction to the more formal aspects of the join calculus can be found in [8], which further discusses its relation to functional programming, surveys its equational theory and proof techniques, and gives operational semantics for concurrency, distribution, and mobility.

2 Concurrent Programming

This section introduces the concurrent and asynchronous aspects of JoCaml, using a series of programming examples. Section 3 will deal with the distributed and mobile aspects. We assume some familiarity with functional programming languages, and in particular Objective Caml.

Notations for Programs. The JoCaml top-level provides an interactive environment, much as the OCaml top-level. Programs consist of a series of top-level statements, terminated by an optional "`;;`" that triggers evaluation in interactive mode. Accordingly, our examples are given as JoCaml statements, followed by the output of the top-level. For instance:

```
# let x = 1  ;;
val x : int
```

```
# print_int (x+1)  ;;
⇒ 2
```

In order to experiment with the examples, you can either type them in a top-level, launched by the command `joctop`, or concatenate program fragments in a source file `a.ml`, compile it with the command `joc -i a.ml` (`-i` enables the output of inferred types), and finally run the program with the command `./a.out`, as performed by the scripts that generate these proceedings.

2.1 Expressions and Processes

JoCaml programs are made of *expressions* and *processes*. Expressions are evaluated synchronously, as usual in functional languages. Indeed, every OCaml expression is also a JoCaml expression.

Processes are executed asynchronously and do not produce any result value, but they can communicate by sending messages on channels (a.k.a. port names). Messages carried by channels are made of zero or more values, which may in turn contain channels.

Simple Channel Declarations. Channels, or *port names*, are the main new primitive values of JoCaml. Port names are either asynchronous or synchronous, depending on their usage for communications: an asynchronous channel is used to send a message; a synchronous channel is used to send a message and wait for an answer.

Channels are introduced using a new `let def` binding, which should not be confused with the ordinary value `let` binding. The right hand-side of the definition of a channel is the process spawned for every message sent on that channel, after substituting the content of the message for the formal parameters on the left hand-side: in short, channels are process abstractions.

For instance, we can define an asynchronous `echo` channel as follows:

```
# let def echo! x = print_int x;  ;;
val echo : <<int>>
```

The new channel `echo` has type `<<int>>`, the type of asynchronous channels carrying values of type `int`. Sending an integer *i* on `echo` fires an instance of the guarded process `print_int i;` which prints the integer on the console. Since `echo` is asynchronous, the sender does not know when the actual printing takes place. Syntactically, the presence of `!` in the definition of the channel indicates that this channel is asynchronous. This indication is present only in the channel definition, not when the channel is used. Also, on the right hand-side, `print_int i` is an expression that returns the unit value `()`, so it is necessary to append a "`;`" to obtain a process that discards this value.

The definition of a synchronous `print` channel is as follows:

```
# let def print x = print_int x; reply  ;;
val print : int -> unit
```

The new channel `print` has type `int -> unit`, the functional type that takes an integer and returns a void result. However, `print` is introduced by `let def` binding (with no `!`), so it is a synchronous channel, and its process on the right hand-side must explicitly send back some (here zero) values as results using a `reply` process. This is an important difference with functions, which implicitly return the value of their main body. Message sending on `print` is synchronous, in the sense that the sender knows that console output has occurred when `print` returns `()`.

Message sending on synchronous channels occurs in expressions, as if they were functions, whereas message sending on asynchronous channels occurs in processes. (The type-checker flags an error whenever a channel is used in the wrong context.) In contrast with value bindings in OCaml, channel definitions always have recursive scopes.

In contrast with traditional channels in process calculi such as CCS, CSP [13], or the pi calculus [22,21], channels and the processes that receive messages on those channels are statically defined in a single `let def` language construct. As a result, channels and functions become quite similar.

Processes. Processes are the new core syntactic class of JoCaml. The most basic process sends a message on an asynchronous channel, such as the channel `echo` defined above. Since only declarations and expressions are allowed at top-level, processes are turned into expressions by "spawning" them : they are introduced by the keyword `spawn` followed by a process in braces "`{ }`".

```
# spawn { echo 1 }  ;;
# spawn { echo 2 }  ;;
⇒ 12
```

Spawned processes run concurrently. The program above may echo 1 and 2 in any order, so the output above may be 12 or 21, depending on the implementation. Concurrent execution may also occur within a process built using the parallel composition operator "`|`". For instance, an equivalent, more concise alternative to the example above is

```
# spawn { echo 1 | echo 2 }  ;;
⇒ 21
```

Composite processes also include conditionals (`if then else`), functional matching (`match with`) and local bindings (`let in` and `let def in`). Process grouping is done by using braces "`{ }`". For instance, the top-level statement

```
# spawn { let x = 1 in
#         { let y = x+1 in echo y | echo (y+1) } | echo x  }  ;;
⇒ 132
```

may output 1, 2, and 3 in any order. Grouping around the process `let y = x+1 in` ... restricts the scope of `y`, so that `echo x` can run independently of the evaluation of y.

Expressions. As usual, expressions run sequentially (in call-by-value) and, when they converge, produce some values. They can occur at top-level, on the right-hand side of value bindings, and as arguments to message sending. Expression grouping is done by using parentheses "`( )`". Apart from OCaml expressions, the most basic expression sends values on a *synchronous* channel and waits for some reply:

```
# let x = 1 in print x ; print (x+1)  ;;
⇒ 12
```

Both expressions `print x` and `print (x+1)` evaluate to the empty result `()`, which can be used for synchronization: the program above always outputs `12`.

Sequences may also occur inside processes. The general form of a sequence inside a process is *expression* `;` *process*, where the result of *expression* will be discarded. As *expression* can itself be a sequence, we can write for instance `spawn { print 1 ; print 2 ; echo 3 }`.

Channels as Values. Channel names are first-class values in JoCaml, which can be locally created, then sent and received in messages. (From a concurrency viewpoint, this is often called *name mobility* [21], and this provides much of the expressiveness to the pi calculus and the join calculus.)

In particular, we can write higher order functions and ports, such as

```
# let async f = let def a! x = f x; in a
# let filter f ch = let def fch! x = ch (f x) in fch
# let def multicast clients =
#   let def mch! x =
#     let cast client = spawn{client x} in
#     { List.iter cast clients; } in
#   reply mch  ;;
val async : ('a -> 'b) -> <<'a>>
val filter : ('c -> 'd) -> <<'d>> -> <<'c>>
val multicast : <<'e>> list -> <<'e>>
```

`async` turns a synchronous channel (or a function) into an asynchronous channel, by discarding its result; `filter f ch` creates a channel that applies `f` to every received message then forwards the result on `ch`; `multicast clients` creates a channel that forwards messages to all `client` channels.

The types for these names and channels are polymorphic: they include type variables such as `'a` that can be replaced with any type. In the example below, for instance, `'a` is instantiated to `string`. (`^` is OCaml string concatenation.)

```
# let echo_string = async print_string
# let tell n = filter (fun x -> x^", "^n^"\n") echo_string
# let yell = multicast [ tell "Cedric"; tell "Fabrice" ]
# ;;
# spawn { yell "Ciao" | yell "Hi" }  ;;
```

```
val echo_string : <<string>>
val tell : string -> <<string>>
val yell : <<string>>
⇒ Hi, Cedric
⇒ Hi, Fabrice
⇒ Ciao, Cedric
⇒ Ciao, Fabrice
```

2.2 Synchronization by Pattern Matching

Join patterns extend port name definitions with synchronization. A pattern defines several ports simultaneously and specifies a synchronization condition to receive messages on these ports. For instance, the following statement defines two synchronizing port names `fruit` and `cake`:

```
# let def  fruit! f | cake! c  =  print_string (f^" "^c^"\n");  ;;
val cake : <<string>>
val fruit : <<string>>
```

To trigger the guarded process `print_string (f^" "^c^"\n");`, messages must be sent on both channels `fruit` and `cake`.

```
# spawn { fruit "apple" | cake "pie" }  ;;
⇒ apple pie
```

The parallel composition operator “`|`” appears both in join-patterns and in processes. This highlights the message combinations consumed by the pattern. The same pattern may be used many times, as long as there are enough messages to consume. When several matches are possible, *which* messages are consumed is left to the implementation.

```
# spawn { fruit "apple" | fruit "raspberry" |
#         cake "pie" | cake "crumble" | cake "jelly" }  ;;
⇒ raspberry pie
⇒ apple crumble
```

Composite join-definitions can also specify several synchronization patterns for the same defined channels.

```
# let def
#    tea! ()    | coin! () = print_string "Here is your tea\n";
# or coffee! () | coin! () = print_string "Here is your coffee\n";
# ;;
# spawn { tea() | coffee() | coin() }  ;;
val coin : <<unit>>
val tea : <<unit>>
val coffee : <<unit>>
⇒ Here is your coffee
```

The name `coin` is defined only once, but can take part in two synchronization patterns. This co-definition is expressed by the keyword `or`. As illustrated here, there may be some internal choice between several possible matches for the same current messages.

Join-patterns are the programming paradigm for concurrency in JoCaml. They allow the encoding of many concurrent data structures. For instance, the following code defines a counter:

```
# let def count! n | inc () = count (n+1) | reply to inc
#      or count! n | get () = count n | reply n to get  ;;
#
# spawn {count 0}  ;;
val inc : unit -> unit
val count : <<int>>
val get : unit -> int
```

This definition calls for two remarks. First, join-pattern may mix synchronous and asynchronous message, but when there are several synchronous message, each `reply` construct must specify the name to which it replies, using the new `reply`...`to` *name* construct. When there is a single synchronous name in the pattern, as in the example above, the `to` construct is optional.

Second, the usage of name `count` is a typical way of ensuring mutual exclusion. For the moment, assume that there is at most one active invocation on `count`. When one invocation is active, `count` holds the counter value as a message and the counter is ready to be incremented or examined. Otherwise, some operation is being performed on the counter and pending operations are postponed until the operation being performed has left the counter in a consistent state. As a consequence, the counter may be used consistently by several threads.

```
# let def wait! t =
#   if get()<3 then wait (t+1) else {print_int t;} in
# spawn { wait 0 | {inc(); inc();} | inc(); }  ;;
⇒ 1
```

Ensuring the correct counter behavior in the example above requires some programming discipline: only one initial invocation on `count` has to be made. If there are more than one simultaneous invocations on `count`, then mutual exclusion is lost. If there is no initial invocation on `count`, then the counter is deadlocked. This can be prevented by making the `count`, `inc` and `get` names local to a `new_counter` definition and then exporting `inc` and `get` while hiding `count`, inside the internal lexical scope of the definition:

```
# let def new_counter () =
#   let def count! n | inc0 () = count (n+1) | reply
#        or count! n | get0 () = count n | reply n in
#   count 0 | reply inc0, get0  ;;
# let inc,get = new_counter ()  ;;
```

```
val new_counter : unit -> (unit -> unit) * (unit -> int)
val inc : unit -> unit
val get : unit -> int
```

This programming style is reminiscent of imperative "object-oriented" programming: a counter is a thing called an object, it has some internal state (`count` and its argument), and it exports some methods to the external world (here, `inc` and `get`). The constructor `new_counter` creates a new object, initializes its internal state, and returns the public methods. Then, a program may allocate and use several counters independently.

2.3 Concurrency Control

Join-pattern synchronization can express many common programming styles, either concurrent or sequential. Next, we give basic examples of abstractions for concurrency.

Synchronization Barriers. A barrier is a common synchronization mechanism. Basically, barriers represent explicit synchronization points, also known as *rendez-vous*, in the execution of parallel tasks.

```
# let def sync1 () | sync2 () = reply to sync1 | reply to sync2 ;;
val sync2 : unit -> unit
val sync1 : unit -> unit
```

The definition includes two `reply` constructs, which makes the mention of a port mandatory. The example below illustrates how the barrier can be used to constrain the interleaving of concurrent tasks. The possible outputs are given by the regular expression $\{12|21\}^*$.

```
# spawn { for i = 0 to 9 do sync1(); print_int 1 done; };
# spawn { for i = 0 to 9 do sync2(); print_int 2 done; }  ;;
⇒ 12121212121212121212
```

Fork/Join Parallelism. Our next example is similar but more general. Consider the sequential function `let evalSeq (f,g) t = (f t, g t)`. We define define a variant, `evalPar`, that performs the two computations `f t` and `g t` in parallel, then joins the two results. We use local channels `cf` and `cg` to collect the results, we spawn an extra process for `cf (f t)`, and evaluate `g t` in the main body of `evalPar`:

```
# let evalPar (f,g) t =
#   let def cf! u | cg v = reply (u,v) in
#   spawn { cf (f t) }; cg (g t)  ;;
#
# let xycoord = evalPar (cos,sin)  ;;
val evalPar : ('a -> 'b) * ('a -> 'c) -> 'a -> 'b * 'c
val xycoord : float -> float * float
```

Bi-directional Channels. Bi-directional channels appear in most process calculi, and in programming languages such as PICT [24] and CML [25]. In the asynchronous pi calculus, for instance, and for a given channel c, a value v can be sent asynchronously on c (written `c![`v`]`) or received from c and bound to some variable x in some guarded process P (written c`?`x`.`P). Any process can send and receive on the channels they know. Finally, the scope of a pi calculus channel name c is defined by the "new" binding $\nu c.P$. In JoCaml, a process can only send messages whereas, for a given name, a unique definition binds the name and receives messages on that name. Nonetheless, bi-directional channels can be defined as follows:

```
# type 'a pi_channel = { snd : <<'a>> ; rcv : unit -> 'a }
# let def new_pi_channel () =
#   let def send! x | receive () = reply x in
#   reply {snd=send; rcv=receive}  ;;
type 'a pi_channel = { snd: <<'a>>; rcv: unit -> 'a }
val new_pi_channel : unit -> 'b pi_channel
```

A pi calculus channel is implemented by a join definition with two port names. The port name `send` is asynchronous and is used to send messages on the channel. Such messages can be received by making a synchronous call to the other port name `receive`. Finally, the new pi calculus channel is packaged as a record of the two new JoCaml names. (Processes and OCaml records both use braces, but in different syntactic contexts.)

Let us now "translate" the pi calculus process

$$\nu c, d.\big(\ c![1] \mid c![5] \mid c?(x).d![x+x] \mid d?(y).print\_int(y)\ \big)$$

We obtain a similar (but more verbose) process:

```
# spawn {
#   let c,d = new_pi_channel(),new_pi_channel() in
#   c.snd 1 | c.snd 5 |
#   {let x = c.rcv() in d.snd (x+x)} |
#   {let y = d.rcv() in print_int y ;} }  ;;
⇒ 2
```

Synchronous pi calculus channels are encoded just as easily as asynchronous ones: it suffices to make `send` synchronous:

```
# type 'a pi_sync_channel = { snd : 'a -> unit; rcv: unit -> 'a }
# let def new_pi_sync_channel () =
#   let def send x | receive () =
#     reply x to receive | reply to send in
#   reply {snd=send; rcv=receive}  ;;
type 'a pi_sync_channel = { snd: 'a -> unit; rcv: unit -> 'a }
val new_pi_sync_channel : unit -> 'b pi_sync_channel
```

2.4 Concurrent Data structures

We continue our exploration of message passing in JoCaml, and now consider some concurrent data structures. (In practice, one would often use the built-in data structures inherited from OCaml rather than their JoCaml internal encodings.)

A Reference Cell. Mutable data structures can be encoded using internal messages that carry the state of the object. A basic example is the imperative variable, also known as reference cell. One method (`get`) examines the content of the cell, while another (`set`) alters it.

```
# type 'a jref = { set: 'a -> unit; get: unit -> 'a }
#
# let def new_ref u =
#   let def state! v | get () = state v | reply v
#        or state! v | set w  = state w | reply   in
#   state u | reply {get=get; set=set}
#
# let r0 = new_ref 0  ;;
type 'a jref = { set: 'a -> unit; get: unit -> 'a }
val new_ref : 'b -> 'b jref
val r0 : int jref
```

Here, the internal state of a cell is its content, its is stored as a message `v` on channel `state`. Lexical scoping is used to keep the state internal to a given cell.

Also, note that the type `'a jref` and `'a pi_sync_channel` are isomorphic; indeed, objects such as mutable references, bi-directional channels, n-place buffers, queues, ... may have the same method interface and implement diverse concurrent behaviors.

A concurrent FIFO. Our second example is more involved. A concurrent FIFO queue is a data structure that provides two methods `put` and `get` to add and retrieve elements from the queue. Unlike a functional queue, however, `get`ting from an empty queue blocks until an element is added, instead of raising an exception.

We give below an implementation that relies (as usual) on two internal lists to store the current values in the queue, but also supports concurrent `get`s and `put`s operations. We use local asynchronous messages to represent the state of the lists, with different messages for empty lists (`inN`, `outN`) and non-empty lists (`inQ`, `outQ`).

– Requests on `put` are always processed at once, using one of the first two patterns, according to the state of the input list.
– Requests on `get` proceed if the output list is non-empty (third pattern)—the auxiliary `outX` channel then returns the head value and updates the state

of the output list. get requests can also proceed if the output list is empty and the input list is non-empty. To this end, the input list is reversed and transferred to the output list.
– There is no pattern for get when both lists are empty, so get requests are implicitly blocked in this case.
– Initially, both lists are empty.

The queue is polymorphic, but its usage is briefly illustrated using integers and series of concurrent puts and gets.

```
# type 'a buffer = { get : unit -> 'a ; put : 'a -> unit }
# let def new_fifo () =
#   let def
#      put i | inN!()   = inQ [i] | reply
#   or put i | inQ! is  = inQ (i::is) | reply
#   or get() | outQ! os = reply outX os
#   or get() | outN!() | inQ! is =
#     inN () | reply outX (List.rev is)
#   or outX os =
#     reply List.hd os | let os' = List.tl os in
#     { if os' = [] then outN() else outQ os' }
#   in
#   inN() | outN() | reply {get=get;put=put}  ;;
#
# let f = new_fifo() in
# spawn { for i = 1 to 9 do f.put i done; };
# spawn { for i = 1 to 5 do print_int (f.get()) done; }  ;;
type 'a buffer = { get: unit -> 'a; put: 'a -> unit }
val new_fifo : unit -> 'b buffer
⇒ 12345
```

2.5 Types and Exceptions

A word on typing. The JoCaml type system is derived from ML and it should be no surprise to functional programmers. In particular, it extends *parametric* polymorphism to the typing of channels. We refer to [9] for a detailed discussion.

Experienced ML programmers may wonder how the JoCaml type system achieves mixing parametric polymorphism and mutable data structures. There is no miracle here. Consider, again, the JoCaml encoding of a reference cell:

```
# let def state! v | get () = state v | reply v
#      or state! v | set w  = state w | reply  ;;
val get : unit -> '_a
val state : <<'_a>>
val set : '_a -> unit
```

The type variable `'_a` that appears inside the types for `state`, `get` and `set` is prefixed by an underscore. Such variables are non-generalized type variables that can be instantiated only once. That is, all the occurrences of `state` must have the same type. Operationally, once `'_a` is instantiated with some type, this type replaces `'_a` in any other types where it occurs (here, the types for `get` and `set`). This guarantees that the various port names whose type contains `'_a` (`state`, `get` and `set` here) are used consistently.

For instance, in the following program, `state 0` and `print_string(get())` force two incompatible instantiations, which leads to a type-checking error (and actually avoids printing an integer while believing it is a string).

```
# let def state! v | get () = state v | reply v
#      or state! v | set w  = state w | reply  ;;
#
# spawn {state 0} ; print_string (get())  ;;
File "ex26.ml", line 6, characters 32-37:
This expression has type int but is here used with type string
```

More generally, whenever the type of several co-defined port names share a type variable, this variable is not generalized. (In ML, the same limitation occurs in the types of identifiers defined by a value binding.) A workaround is to encapsulate the definition into another one, which gives another opportunity to generalize type variables:

```
# let def new_ref v =
#   let def state! v | get () = state v | reply v
#        or state! v | set w  = state w | reply
#   in spawn {state v} ; reply (get,set)  ;;
val new_ref : 'a -> (unit -> 'a) * ('a -> unit)
```

Exceptions. Exceptions and exception handling within expressions behave as in OCaml. If an exception is not caught in the current expression, however, its handling depends on the synchrony of the process.

If the process is asynchronous, the exception is printed on the standard output and the asynchronous process terminates. No other process is affected.

```
# spawn { failwith "Bye"; }; print_string "Done"  ;;
⇒ Uncaught exception: Failure("Bye")
⇒ Done
```

If the process is synchronous, every joint call terminates with the exception instead of a reply. In particular, when a pattern contains several synchronous channels, the exception is replicated and thrown to all blocked callers:

```
# let catch x = try x() with Failure s -> print_string s in
# let def a () | b () =
#   failwith "Bye "; reply to a | reply to b in
# spawn { {catch a;} | {catch b;} }  ;;
⇒ Bye Bye
```

Exercise 1. The "core join calculus" consists only of asynchronous channels and processes. Sketch an encoding of synchronous channels and expressions into this subset of JoCaml. (Hint: this essentially amounts to a call-by-value continuation-passing encoding.)

Exercise 2 (Fairness). What kind of fairness is actually provided by JoCaml when several messages are available on the same channel? When different patterns could be used for the same messages? Try to define stronger fairness properties and to implement them for some examples of join patterns.

3 Distributed Programming

JoCaml has been designed to provide a simple and well-defined model of distributed programming. Since the language entirely relies on asynchronous message passing, programs can either be used on a single machine (as described in the previous section), or they can be executed in a distributed manner on several machines.

In this section, we give a more explicit account of distribution. We describe support for execution on several machines and new primitives that control locality, migration, and failure. To this end, we interleave a description of the model with a series of examples that illustrate the use of these primitives.

3.1 The Distributed Model

The execution of JoCaml programs can be distributed among several machines, possibly running different systems; new machines may join or quit the computation. At any time, every process or expression is running on a given machine. However, they may migrate from one machine to another, under the control of the language. In this implementation, the runtime support consists of several system-level processes that communicate using TCP/IP over the network.

In JoCaml, the execution of a process (or an expression) does not usually depend on its localization. Indeed, it is equivalent to run processes `P` and `Q` on two different machines, or to run the compound process `{ P | Q }` on a single machine. In particular, the scope for defined names and values does not depend on their localization: whenever a port name appears in a process, it can be used to form messages (using the name as the address, or as the message content) without knowing whether this port name is locally- or remotely-defined, and which machine will actually handle the message. As a first approximation, locality is transparent, and programs can be written independently of their run-time distribution.

Of course, locality matters in some circumstances: side-effects such as printing values on the local console depend on the current machine; besides, efficiency can be affected because message sending over the network takes much longer than local calls; finally, the termination of some underlying runtime will affect all its local processes. For these reasons, locality is explicitly controlled by the programmer, and can be adjusted using migration. Conversely, resources such as definitions and processes are never silently relocated by the system—the programmer interested in dynamic load-balancing must code relocation as part of his application.

An important issue when passing messages in a distributed system is whether the message content is copied or passed by reference. This is the essential difference between functions and synchronous channels.

– When a function is sent to a remote machine, a copy of its code and values for its local variables are also sent there. Afterwards, any invocation will be executed locally on the remote machine.
– When a synchronous port name is sent to a remote machine, only the name is sent (with adequate routing information) and invocations on this name will forward the invocation to the machine where the name is defined, much as in a remote procedure call.

The name-server. Since JoCaml has lexical scoping, programs being executed on different runtimes do not initially share any port name; therefore, they would normally not be able to interact with one another. To bootstrap a distributed computation, it is necessary to exchange a few names, and this is achieved using a built-in library called the name server. Once this is done, these first names can be used to communicate some more values (and in particular port names) and to build more complex communication patterns.

The interface of the name server mostly consists of two functions to register and look up arbitrary values in a “global table” indexed by plain strings. Pragmatically, when a JoCaml program (or top-level) is started, it takes as parameters the IP address and port number of a name server. The name server itself can be launched using the command `jocns`.

The following program illustrates the use of the name server, with two processes running in parallel (although still in the same runtime). One of them locally defines some resource (a function `f` that squares integers) and registers it under the string `square`. The other process is not within the scope of `f`; it looks up for the value registered under the same string, locally binds it to `sqr`, then uses it to print something.

```
# spawn{ let def f x = reply x*x
#        in Ns.register "square" f vartype; };;
#
# spawn{ let sqr = Ns.lookup "square" vartype
#        in  print_int (sqr 2); };;
Warning: VARTYPE replaced by type ( int ->  int) metatype
Warning: VARTYPE replaced by type ( int ->  int) metatype
⇒ 4
```

The vartype keyword stands for the (runtime representation of the) type of the value that is being registered or looked up, which is automatically inserted by the compiler. When a value is registered, its type is explicitly stored with it. When a value is looked up, the stored type is compared with the inferred type in the receiving context; if these types do not match, an exception TypeMismatch is raised. This limited form of dynamic typing is necessary to ensure type safety. To prevent (and explain) runtime TypeMismatch exceptions, the compiler also issues a warning that provides the inferred vartype at both ends of the name server, here int -> int. (When writing distributed program fragments, it is usually a good idea to share the type declarations in a single .mli file and to explicitly write these types when calling the name server.)

Of course, using the name server makes sense only when the two processes are running as part of stand-alone programs on different machines, and when these processes use the same conventional strings to access the name server. To avoid name clashes when using the same name server for unrelated computations, the indexed string is prefixed by a local identifier Ns.user; by default, Ns.user contains the local user name.

3.2 Locations and Mobility

So far, the localization of processes and expressions is entirely static. In some cases, a more flexible control is called for. Assume that "square computations" are best performed only on the server machine that exports the square port, and that a client machine needs to compute sums of squares. If the client uses a loop to compute the sum by remote calls on square, each call within the loop would result in two messages on the network (one for the request, and another one for the answer). It would be better to run the loop on the machine that actually computes the squares. Yet, we would prefer not to modify the program running on the server every time we need to run a different kind of loop that involves numerous squares.

To this end, we introduce a unit of locality called "location". A location contains a bunch of definitions and running processes "at the same place". Every location is given a name, and these location names are first-class values. They can be communicated as content of messages and registered to the name server. These location names can also be used as arguments to primitives that dynamically control the relations between locations.

Basic examples. Locations can be declared either locally or as a top-level statement. For instance, we create a new location named here:

```
# let loc here
#   def square x = reply x*x
#   and cubic x = reply (square x)*x
#   do { print_int (square 2); }
# ;;
# print_int (cubic 2)
val here : Join.location
val cubic : int -> int
val square : int -> int
⇒ 48
```

This `let loc` declaration binds a location name `here` and two port names `square` and `cubic` whose scope extends to the location and to the following statements. Here, the location also has an initial process `print_int (square 2);` introduced by `do {}` (much like `spawn {}` in expressions). This process runs within the location, in parallel with the remaining part of the program. As a result, we can obtain either `84` or `48`.

Distributed computations are organized as trees of nested locations; every definition and every process is permanently attached to the location where it appears in the source program. Since `let loc`s can occur under guards, processes and expressions can create new locations at runtime, with their initial content (bindings and processes) and a fresh location name. The new location is placed as a sub-location of the location that encloses the `let loc`. Once created, there is no way to add new bindings and processes to the location from outside the location.

For instance, the following program defines three locations such that the locations named `kitchen` and `living_room` are sub-locations of `house`. As regards the scopes of names, the locations `kitchen`, `living_room` and the ports `cook`, `switch`, `on`, `off` all have the same scope, which extends to the whole `house` location (between the first `do {` and the last `}`). Only the location name `house` is visible from the rest of the source file.

```
# let loc house do {
#  let loc kitchen
#   def cook() = print_string " Cooking... "; reply
#   do {}
#  and living_room
#   def switch()| off!() = print_string "Music on."; reply | on()
#   or  switch()| on!() = print_string "Music off."; reply | off()
#   do { off() }
#  in
#  switch(); cook(); switch(); }
val house : Join.location
⇒ Music on. Cooking... Music off.
```

Mobile Agents. While processes and definitions are statically attached to their location, locations can move from one enclosing location to another. Such migrations are triggered by a process inside of the moving location (a "subjective move", in Cardelli's terminology [5]). As a result of the migration, the moving location becomes a sub-location of its target location. Note that locations can be used for several purposes: as destination addresses, as mobile agents, or as a combination of the two.

Our next example is an agent-based program to compute a sum of squares. On the server side, we create an empty location, `here`, and we register it on the name-server; its name will be used as the target address for our mobile agent.

```
# let loc here do {} in Ns.register "here" here vartype
# ;;
# Join.server()
```

(The call to `Join.server()` prevents the immediate termination of the JoCaml runtime, even if it has no active process or expression: further local activity can occur later, as the result of remote messages and migrations.)

On the client side, we create another location, `mobile`, that wraps the loop computation that should be executed on the square side; the process within `mobile` first looks up the name `here`, then moves itself inside of "`here`", and finally performs the computation.

```
# let loc mobile
#   do {
#     let there = Ns.lookup "here" vartype in
#     go there;
#     let sqr = Ns.lookup "square" vartype in
#     let def sum (s,n) =
#       reply (if n = 0 then s else sum (s+sqr n, n-1)) in
#     print_string (sum(0,5));
#   }
```

The `go there` expression migrates the `mobile` location with its current content to the server machine, as a sub-location of location `here`, then completes and returns `()`. Afterwards, the whole computation (calls to the name server, to `sqr` and to `sum`) is local to the server. There are only three messages exchanged between the machines: one for `Ns.lookup`, one for the answer, and one for the migration.

Applets. The next example shows how to define applets. An applet is a program that is downloaded from a remote server, then used locally. As compared to the previous examples, migration operates the other way round, from the server to the client. For our purposes, the applet implements a mutable cell with destructive reading:

```
# let def new_cell there =
#   let def log s = print_string ("cell "^s^"\n"); reply in
#   let loc applet
#     def get() | some! x = log ("is empty"); none() | reply x
#     and put x | none!() = log ("contains "^x); some x | reply
#     do { go there; none () } in
#   reply get,put
# ;;
# Ns.register "cell" new_cell vartype;
# Join.server
```

Our applet has two states: either `none()`or `some s` where `s` is a string, and two methods `get` and `put`. Each time `cell` is called, it creates a new applet in its own location. Thus, numerous independent cells can be created and shipped to callers. The name `cell` takes as argument the location (`there`) where the new cell should reside. The relocation is controlled by the process `go there; none ()` that first performs the migration, then sends an internal message to activate the cell. Besides, `cell` defines a `log` function outside of the applet. The latter therefore remains on the server and, when called from within the applet on the client machine, keeps track of the usage of its cell. This is in contrast with applets à la Java: the location migrates with its code, but also with its communication capabilities unaffected.

We supplement our example with a basic user that allocates and uses a local cell:

```
# let cell = Ns.lookup "cell" vartype
#
# let loc user
#   do {
#     let get, (put : string -> unit) = cell user in
#     put "world";
#     put ("Hello, "^get ());
#     print_string (get ());
#   }
```

On the client machine, we observe "`Hello, world`" on the console, as could be expected. Besides, on the server side, we observe the log:

```
⇒ cell is empty
⇒ cell contains world
⇒ cell is empty
⇒ cell contains Hello, world
⇒ cell is empty
```

On the client machine, there are no more `go` primitives in the applet after its arrival, and this instance of the location name `applet` does not appear anywhere. As a result, the contents of the applet can be considered part of the host location, as if this content had been defined locally in the beginning. (Some other host location may still move, but then it would carry the cell applet as a sub-location.)

Exercise 3 (Local State). What is the experimental distributed semantics of mutable references? What about global references and modules? Write a function that allocates a "correct" distributed reference with an interface for reading, writing, and relocating the reference.

3.3 Termination, Failures, and Failure Recovery

As a matter of fact, some parts of a distributed computation may fail (e.g., because a machine is abruptly switched off). The simplest solution would be to abort the whole computation whenever this is detected, but this is not realistic in case numerous machines are involved. Rather, we would like our programs to detect such failures and take adequate measures, such as cleanly report the problem, abort related parts of the computation, or make another attempt on a different machine. To this end, JoCaml provides an abstract model of failure and failure detection expressed in terms of locations:

- a location can run a primitive process `halt()` that, when executed, atomically halts every process inside of this location (and recursively every sub-location);
- a location can detect that another location with name `there` has halted, using a primitive expression of the form `fail there;`P. The expression blocks, until the failure of location `there` is detected. When the process P runs, it is guaranteed that location `there` is halted for any other location trying to access `there`.

The `halt` primitive can be seen as a way to reflect, in the model, the abrupt failure of a machine that hosts the `halt`ed locations. For instance, a fallible machine running a process P can be seen as a top-level location

```
let loc runtime do { P | halt() }
```

Since locations fail only as a whole, the programmer can define locations as suitable units of failure recovery, pass their names to set up remote failure detection, and even use `halt` and `fail` primitives to control the computation. By design, however, no silent recovery mechanism is provided: the programmer must figure out what to do in case of partial failure.

The `fail` primitive is tricky to implement (it cannot be fully implemented on top of an asynchronous network, for instance). On the other hand, it does provide the expected negative guarantees: the failed location is not visible anymore, from any part of the computation, on any machine. In the current implementation, halting is detected only when (1) the `halt ()` primitive is issued in the same runtime as the `fail`, or (2) the JoCaml runtime containing the location actually stops. (Thus, simply running `halt ()` does not trigger matching `fail`s in other runtimes, but `exit 0;` will trigger them.)

A Computation Supervisor. There is usually no need to halt locations that completed their task explicitly (the garbage-collector should take care of them). However, in some case we would like to be sure that no immigrant location is still running locally.

Let us assume that `job` is a remote function within location `there` that may create mobile sub-locations and migrate them to the caller's site. To this end, the caller should supply a host location, as in the previous examples. How can we make sure that `job` is not using this location to run other agents after the call completes ? This is handled using a new temporary location `box` for each call, and halting it once the function call has completed.

```
# let def safe! (job,arg,success,failure) =
#   let loc box
#     def  kill!() = halt();
#     and  start() = reply job (box,arg) in
#   let
#     def got! x  | live!()   = got x | kill()
#     or  got! x  | halted!() = success x
#     or  live!() | halted!() = failure () in
#   got (start()) | live() | fail box; halted()
val safe :
  <<((Join.location * 'a -> 'b) * 'a * <<'b>> * <<unit>>)>>
```

Our supervising protocol either send a result on `success` or a signal on `failure`. In both cases, the message guarantees that no alien computation may take place afterward on the local machine.

The protocol consists of a host location and a supervisor definition. Initially, there is a `live()` message and the supervisor waits for either a result on `got` or some failure report on `halted`. Depending on the definition of `job`, the expression `job(box,arg)` can create and move locations inside of the box, communicate with the outside, and eventually reply some value within the box. Once this occurs, `got` forwards the reply to the control process, and the first join-pattern is triggered. In this case, the `live()` message is consumed and eventually replaced by a `halted()` message (once the `kill()` message is handled, the box gets halted, and the fail guard in the control process is triggered, releasing a message on `halted`).

At this stage, we know for sure that no binding or computation introduced by `job` remains on the caller's machine, and we can return the value as if a plain RPC had occurred.

This "wrapper" is quite general. Once a location-passing convention is chosen, the `safe` function does not depend on the actual computation performed by `job` (its arguments, its results, and even the way it uses locations are parametric here). We could further refine this example to transform unduly long calls to `job` into failure (by sending a `kill ()` message after an external timeout), and to delegate some more control to the caller (by returning kill at once).

Exercise 4 (Mobility). Starting from your favorite functional program, add locations and mobility control to distribute the program on several machines, and speed up the computation.

A Concurrent Programming (Lab Session)

We suggest a series of exercices to experiment with asynchronous message passing in JoCaml, including classic programming examples. We also provide some solutions. One may also begin with the examples in the previous sections, or even try to implement one's favorite concurrent algorithm.

Exercise 5 (Fibonacci). Assume we are computing the Fibonacci series on values with a slow (but parallel) addition, rather than integers. For example:

```
# type slow = int
#
# let delay = ref 0.1
# let add (a:slow) (b:slow) =
#   Thread.delay !delay; (a+b : slow)
#
# let rec fib = function
#   | 0 -> 0 | 1 -> 1 | n -> add (fib (n-1)) (fib(n-2))  ;;
type slow = int
val delay : float ref
val add : slow -> slow -> slow
val fib : int -> slow
```

Write a faster, parallel version of `fib`. What kind of speedup should we obtain? What is actually observed? Does that depend on `delay`?

Exercise 6 (Locks). Write a JoCaml implementation of locks, with the following interfaces:

1. basic lock, with a synchronous `get` channel to acquire the lock and an asynchronous `release` channel to release the lock.
2. n-user lock, with the same interface but up to n concurrent holders for the lock.
3. reader-writer locks, with interface
   - `acquire_shared` to get a non-exclusive lock (or block until available),
   - `release_shared` to release the non-exclusive lock,
   - `acquire_exclusive` to get an exclusive lock (or block until available),
   - `release_exclusive` to release it.
4. reader-writer locks with fairness between writers and readers: provided all locks are eventually released, any `acquire` request is eventually granted.

Solutions

Fibonacci (Exercise 5). We can use fork/join parallelism, e.g.

```
# let rec pfib = function
#   | 0 -> 0 | 1 -> 1 | n ->
#   let def a! v | b u = { reply (add u v) } in
#   spawn {a (pfib (n-2))}; b(pfib(n-1))
val pfib : int -> Ex1.slow
```

We obtain (on a laptop running Windows XP):

```
# let time f v =
#   let t = Unix.gettimeofday () in let r = f v in
#   let t' = Unix.gettimeofday () in t' -. t in
# let test size =
#   let t0,t1 = time fib size, time pfib size in
#   Printf.printf
#     "delay=%1.1e size=%2d base=%4.2f fj=%4.2f speedup=%f"
#     !delay size t0 t1 ((t0-. t1)/. t0) ; print_newline() in
# delay:= 0.001        ; test 12;
# delay:= !delay /. 10.; test 16;
# delay:= !delay /. 10.; test 17;
# delay:= !delay /. 10.; test 18;
# delay:= !delay /. 10.; test 19;
# delay:= !delay /. 10.; test 19;
# delay:= !delay /. 10.; test 20;
⇒ delay=1.0e-03 size=12 base=0.54 fj=0.02 speedup=0.953618
⇒ delay=1.0e-04 size=16 base=3.73 fj=0.08 speedup=0.978815
⇒ delay=1.0e-05 size=17 base=5.53 fj=0.13 speedup=0.975786
⇒ delay=1.0e-06 size=18 base=4.61 fj=0.22 speedup=0.951420
⇒ delay=1.0e-07 size=19 base=7.57 fj=0.45 speedup=0.940275
⇒ delay=1.0e-08 size=19 base=0.19 fj=0.38 speedup=-1.059140
⇒ delay=1.0e-09 size=20 base=0.31 fj=0.91 speedup=-1.999999
```

Locks (Exercise 6).

```
# type lock = { acquire : unit -> unit ; release : <<unit>> }  ;;
#
# let new_lock() =
#   let def acquire() | release!() = reply in
#   spawn{ release() };
#   {acquire=acquire; release=release}  ;;
#
# let new_nlock n =
#   let def acquire() | token!() = reply in
#   for i = 1 to n do spawn{ token() } done;
#   {acquire=acquire; release=token}  ;;
#
# let new_rwlock() =
#   let def
#      acquire_exclusive() | idle!()    = reply
#   or acquire_shared()    | idle!()    = shared 1 | reply
#   or acquire_shared()    | shared! n  = shared (n+1) | reply
#   or release_shared!()   | shared! n  =
#      if n==1 then idle() else shared (n-1) in
#   spawn { idle() };
#   {acquire=acquire_shared; release=release_shared},
#   {acquire=acquire_exclusive; release=idle}  ;;
#
# let new_rwfairlock() =
#   let def
#      acquire_exclusive() | idle!() = reply
#   or acquire_shared()    | idle!()    = shared 1 | reply
#   or acquire_shared()    | shared! n  = shared (n+1) | reply
#   or release_shared!()   | shared! n  =
#      if n==1 then idle() else shared (n-1)
#
#   or acquire_exclusive() | shared! n = waiting n | reply wait()
#   or release_shared!()   | waiting! n =
#      if n==1 then ready() else waiting (n-1)
#   or wait() | ready!() = reply in
#   spawn { idle() };
#   {acquire=acquire_shared; release=release_shared},
#   {acquire=acquire_exclusive; release=idle}  ;;
type lock = { acquire: unit -> unit; release: <<unit>> }
val new_lock : unit -> lock
val new_nlock : int -> lock
val new_rwlock : unit -> lock * lock
val new_rwfairlock : unit -> lock * lock
```

In all these examples, we rely on the caller to enforce the lock discipline: only release once, after acquiring the lock. We could also provide abstractions to enforce it, e.g.

```
# let synchronized lock job v =
#   lock.acquire(); let r = job v in spawn{ lock.release() }; r ;;
val synchronized : Ex4.lock -> ('a -> 'b) -> 'a -> 'b
```

B Distributed and Mobile Programming (Lab Session)

Exercise 7. Use the nameserver to exchange a simple string between two runtimes: write a first program to register your name using the string "name", and a second one to lookup the string "name", and print it on the terminal.

Exercise 8 (Remote Shell Command).

1. Write a program that registers a synchronous channel on the nameserver. The channel takes a string as argument, calls `Sys.command` to execute it, and returns the error code for the command.
   Write a second program to lookup this channel and execute some commands remotely.

2. Write a program that registers a synchronous channel on the nameserver. The channel returns a new location that can be used to send an agent on the computer.
   Write a second program that sends an agent to the location, lists all files in the "/tmp" directory (use `Unix.opendir`, `Unix.readdir`[raises an exception at end] and `Unix.closedir`), and returns the list on the caller machine.

Exercise 9. Write a "chat" program with JoCaml:

1. Write a channel of type: `<<string * string>>`—the first string is a user name, the second string is a message from that user. Register the channel on the name server under your name (use `Ns.user := "pub"` in your program for all users to be able to access your name).
2. Write a function that sends a message to a friend: the function should lookup a channel on the nameserver from the friend name, then send a message on this channel.
3. Add *chat rooms* to your program:
   - Write a chat rooms server, that will manage all the chat rooms.
   - Write different programs:
     - To list all the existing chat rooms
     - To add a new chat room
     - To join a chat room: this client should be able to read the user input (use the function `read_line` for this), send the message to the chat room, and display messages received from other users of the chat room.

Name Server (Exercise 7). First program:

```
Ns.register "name" "My Name" vartype;;
```

Second program:

```
let name = Ns.lookup "name" vartype;;
print_string name; print_newline ();;
```

RSH (Exercise 8.1). First program:

```
let def rsh(command) = reply (Sys.command command);;
Ns.register "my computer name" rsh vartype;;
```

Second program:

```
let rsh = Ns.lookup "my computer name" vartype  ;;
print_int (rsh "xterm"); print_newline ()  ;;
```

RSH (Exercise 8.2). First program:

```
let def rsh_loc () =
  let loc new_location do {} in
  reply new_location  ;;

Ns.register "my computer name" rsh_loc vartype  ;;
```

Second program:

```
let (rsh_loc : unit -> Join.location) =
  Ns.lookup "my computer name" vartype ;;

let def print_list! list =
  { List.iter (fun s -> print_string s; print_newline()) list; };;

let loc listdir_agent do
  { Join.go (rsh_loc ());
    let list = ref [] in
    (let dir = Unix.opendir "/tmp" in
     try
       while true do
         list := (Unix.readdir dir) :: !list
       done
     with _ -> Unix.closedir dir);
     print_list !list
  };;
```

Chat (Exercise 9). A complete implementation can be found at
http://pauillac.inria.fr/jocaml/afp2002/chat.ml.

Acknowledgement.. Many thanks to the members of the Moscova project at INRIA, and in particular Georges Gonthier, James J. Leifer, and Jean-Jacques Lévy.

References

1. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 2000.
2. D. S. Alexander, M. W. Hicks, P. Kakkar, A. D. Keromytis, M. Shaw, J. T. Moore, C. A. Gunter, T. Jim, S. M. Nettles, and J. M. Smith. The switchware active network implementation. In *The ML Workshop*, International Conference on Functional Programming (ICFP), Sept. 1998. ACM SIGPLAN.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In B. Magnusson, editor, *ECOOP 2002 – Object Oriented Programming*, volume 2374 of *LNCS*, pages 415–440. Springer-Verlag, jun 2002.

4. A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. Research Report 20, DEC SRC, Aug. 1987.
5. L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.
6. S. Conchon and F. Le Fessant. Jocaml: Mobile agents for objective-caml. In *ASA/MA'99*, pages 22–29. IEEE Computer Society, Oct. 1999.
7. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, Jan. 1996.
8. C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics. International Summer School, APPSEM 2000, Caminha, Portugal, Sept. 2000*, volume 2395 of *LNCS*, pages 268–332. Springer-Verlag, 2002. Also available from `http://research.microsoft.com/~fournet`.
9. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *8th International Conference on Concurrency Theory*, volume 1243 of *LNCS*, pages 196–212. Springer-Verlag, July 1997.
10. C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Proceedings of IFIP TCS 2000*, volume 1872 of *LNCS*. IFIP TC1, Springer-Verlag, Aug. 2000.
11. R. S. Gray. Agent tcl: A transportable agent system. In *CIKM Workshop on Intelligent Information Agents*, Baltimore, Maryland, dec 1995. Fourth International Conference on Information and Knowledge Management (CIKM 95).
12. M. Hayden. Distributed communication in ML. Technical Report TR97-1652, Cornell University, Computer Science, Nov. 11, 1997.
13. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
14. D. Johansen, R. V. Renesse, and F. B. Schneider. An introduction to the tacoma distributed system version 1.0. Technical Report 95-23, University of TromsO, Norway, June 1995.
15. F. Le Fessant. The JoCAML system prototype (beta 1.08). Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998–2002.
16. F. Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Principles of Distributed Computing (PODC)*, Rhodes Island, Aug. 2001.
17. F. Le Fessant and L. Maranget. Compiling join-patterns. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, Sept. 1998.
18. F. Le Fessant, I. Piumarta, and M. Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, Montreal (Canada), June 1998. ACM SIGPLAN.
19. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition edition, 1999.
20. X. Leroy and al. The Objective CAML system 3.05. Software and documentation available from `http://caml.inria.fr`.
21. R. Milner. *Communication and Mobile Systems: the π-Calculus*. Cambridge University Press, Cambridge, 1999.
22. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, Sept. 1992.

23. M. Odersky. Functional nets. In *Proceedings of the European Symposium on Programming*, volume 1782 of *LNCS*, pages 1–25. Springer Verlag, 2000.
24. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, May 2000.
25. J. H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, pages 165–198. Springer-Verlag, 1992.
26. F. Rouaix. A web navigator with applets in caml. In *Fifth WWW Conference*, Paris, May 1996.
27. A. Schmitt. Safe dynamic binding in the join calculus. In *IFIP TCS'02*, Montreal, Canada, 2002.
28. A. Schmitt and J.-B. Stefani. The M-calculus: A higher order distributed process calculus. In *Proceeding 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*. ACM, 2003.
29. J. White. Telescript technology: Mobile agents. In *Software Agents*. J. Bradshaw, editor, AAAI Press/MIT Press, 1996.