The JoCaml language
Release 3.11

Documentation and user's manual

Louis Mandel and Luc Maranget

December 12, 2008

Contents

Foreword

This manual documents the release 3.11 of the JoCaml system. JoCaml is an extension of Objective Caml.

License

As an extension, JoCaml includes much source code from Objective Caml. It should be no surprise that JoCaml license is exactly Objective Caml license.

The JoCaml system is open source and can be freely redistributed. See the file `LICENSE` in the distribution for licensing information.

Availability

The complete JoCaml distribution can be accessed via the Web site `http://jocaml.inria.fr/`. This Web site contains some additional information on JoCaml.

Acknowledgements

We thank the whole Objective Caml development team, and in particular Xavier Leroy, for giving us full access to source code, computer resources, names, logos etc. All bugs we have introduced are ours.

JoCaml and this manual owe much to previous work, by numerous people, including Fabrice Le Fessant, Cédric Fournet and Alan Schmitt.

The software authors are Luc Maranget, Ma Qin and Louis Mandel.

Part I

An introduction to JoCaml

Chapter 1

Concurrent programming

This part of the manual is a tutorial introduction to JoCaml. This chapter starts with small, local examples. Then it deals with the distributed features. It is assumed that the reader has some previous knowledge of Objective Caml.

1.1 Conventions

Examples are given as JoCaml source, followed by the output of the top-level (or of the compiler when prompted to print types). The JoCaml top-level provides an interactive environment, much as the Objective Caml top-level.

In order to try the examples, you can either type them in a top-level, launched by the command `jocaml`, or concatenate the sources chunks in some file `a.ml` then compile `a.ml` by the command `jocamlc a.ml`, and finally run the produced code by the command `./a.out`.

1.2 Basics

JoCaml programs are made of *processes* and *expressions*. Roughly, processes are executed asynchronously and produce no result, whereas expressions are evaluated synchronously and their evaluation produces values. For instance, Objective Caml expressions are JoCaml expressions. Processes communicate by sending messages on channels (a.k.a. port names). Messages carried by channels are made of zero or more values, and channels are values themselves. In contrast with other process calculi (such as the pi-calculus and its derived programming language Pict), channels and the processes that listen on them are defined in a single language construct. This allows considering (and implementing) channels as functions when they have the same usage.

JoCaml programs are first organized as a list of top-level statements. A Top-level statement is a declaration (such as an Objective Caml binding `let x = 1` or a channel binding) or an expression. Top-level statements are terminated by an optional `;;` that triggers evaluation in interactive mode.

1.2.1 Simple channel declarations

Channels, or *port names*, are the main new primitive values of JoCaml.

Users can create new channels with a new kind of `def` binding, which should not be confused with the ordinary value `let` binding. The right hand-side of the definition of a channel a is a

process that will be spawned whenever some message is sent on a. Additionally, the contents of messages received on a are bound to formal parameters.

For instance, here is the definition of an `echo` channel:

```
# def echo(x) = print_int x; 0
# ;;
val echo : int Join.chan = <abstr>
```

The new channel `echo` has type `int Join.chan`, which is the type of channels carrying values of type `int`. Sending an integer i on `echo` fires an instance of the guarded process `print_int x; 0` which prints the integer on the console. Since the Objective Caml expression `print_int x` returns the value `()`, it is necessary to append a `; 0` that discards this value. `0` is the empty process. As a regards syntax, the parenthesis around the formal argument "`x`" are mandatory.

Channel `echo` is an *asynchronous* channel since sending a message on this channel is a non-blocking operation and it is not possible to know when the actual printing takes place.

1.2.2 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` just introduced. 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`".

```
# spawn echo(1)
# ;;
- : unit = ()
# spawn echo(2)
# ;;
- : unit = ()
12
```

Processes introduced by "`spawn`" are executed concurrently. The program above may either echo `1` then `2`, or echo `2` then `1`. Thus, the output above may be `12` or `21`, depending on the implementation. The processes `echo(1)` and `echo(2)` are examples of the the most basic process: sending a message on some channel. The syntax for such message sending is the same as the one for function call in Objective Caml. Hence, writing `echo 1` and `echo 2` is correct. However, in this manual, we conventionally write message sending with argument between parenthesis.

Concurrent execution also occurs inside processes, using the parallel composition operator "`&`". This provides a more concise, semantically equivalent, alternative to the previous example:

```
# spawn echo(1) & echo(2)
# ;;
- : unit = ()
12
```

Composite processes also include conditionals (`if`'s), matching (`match`'s) and local binding (`let...in`'s and `def...in`'s). Process grouping is done by using brackets "`(`" and "`)`" or the equivalent "`begin`" and "`end`", just as in Objective Caml.

```
# spawn begin
#   let x = 1 in
#   (let y = x + 1 in echo(y) & echo(y+1)) & echo(x)
# end
# ;;
- : unit = ()
231
```

Once again, the program above may echo the integers `1`, `2` and `3` in any order. Grouping is necessary around the process `let y = ...in` to restrict the scope of `y` such that its evaluation occurs independently of the process `echo(x)`.

Processes may also include sequences. 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, thus one may write:

```
# spawn begin
#   print_int 1; print_int 2; echo(3)
# end
# ;;
- : unit = ()
123
```

A sequence may be terminated by an empty process that does nothing and is denoted by "`0`". Thus, an alternative to the previous example is as follows:

```
# spawn begin
#   print_int 1; print_int 2; print_int 3; 0
# end
# ;;
- : unit = ()
123
```

This is why `print_int x; 0` in the definition of the `echo` channel is considered as a process.

1.2.3 More on channels

The guarded process in a channel definition can spawn several messages, as in a stuttering `echo` channel:

```
# def echo_twice(x) = echo(x) & echo(x)
# ;;
val echo_twice : int Join.chan = <abstr>
```

It is also possible to define directly such a channel, without referring to the channel `echo`, but by using the Objective Caml function `print_int`. In this case, it is necessary to enclose each use of `print_int` in "`(`" and "`)`", as in this new definition of `echo_twice`:

```
# def echo_twice(x) = (print_int x; 0) & (print_int x; 0)
# ;;
val echo_twice : int Join.chan = <abstr>
```

Such grouping is necessary because "&" binds more tightly than ";", as in:

```
# def echo3(x) = print_int x; echo(x) & echo(x)
# ;;
val echo3 : int Join.chan = <abstr>
```

Channels may accept tuple of values as arguments, and those arguments can be destructured with pattern matching notation. For example, the following channel `f` accepts pairs as shown by its type.

```
# def strange_echo(x,y) = echo (x+y) & echo (y-x)
# ;;
val strange_echo : (int * int) Join.chan = <abstr>
```

Hence, in JoCaml, polyadic channels are simply expressed as (monadic) channels that carry tuples.

Port names are first-class values in JoCaml. They can be sent as messages on other port names. As a result, higher order "ports" can be written, such as

```
# def twice(f,x) = f(x) & f(x )
# ;;
val twice : ('a Join.chan * 'a) Join.chan = <abstr>
```

The type for `twice` is polymorphic: it includes a type variable `'a` that can be replaced by any type. Thus `twice` is a channel that takes a channel of type `'a Join.chan` and a value of type `'a` as arguments.

For instance, `'a` can be the type of integers or the type of strings:

```
# def echo_string(s) = print_string s; 0
# ;;
val echo_string : string Join.chan = <abstr>
# spawn twice(echo,0) & twice(echo_string,"X")
# ;;
- : unit = ()
00XX
```

1.2.4 Synchronous channels

One perfectly can have a process to "return a value": it suffices to parameterize it with a continuation.

```
# def succ(x,k) =  print_int x; k(x+1)
# ;;
val succ : (int * int Join.chan) Join.chan = <abstr>
```

Here, `succ` prints the value of `x`, and *then* sends the message `x+1` to its continuation `k`. We insist on *then* : when the message is received by whoever is waiting at the other end, we can be sure that `x` has been printed. Or, more precisely, if the receiver also prints something, "something" should appear on the console after `x`. Let us define a continuation for `succ`:

```
# def k(x) = succ(x,echo)
# ;;
val k : int Join.chan = <abstr>
# spawn succ(1,k)
# ;;
- : unit = ()
123
```

And we have yet another example of printing `123` in that order.

Although it can be fun, continuation passing style is not very convenient. JoCaml provides *synchronous* channels to define processes that return values more directly. The previous example can be written as follows:

```
# def succ(x) = print_int x; reply x+1 to succ
# ;;
val succ : int -> int = <fun>
```

The type of `succ` is the functional type `int -> int` that takes one integer as argument and returns an integer. However, `succ` is not a function because it is introduced by a `def` binding. Since the process terminates with `reply x+1 to succ`, `succ` is a synchronous channel which returns the `x+1` value. The mechanism to return values for synchronous channels is different from the one for functions: it uses a `reply/to` construct whose semantics is to send back some values as result. This is the first difference with plain Objective Caml functions, which implicitly return the value of the guarded expression, instead of using the explicit `reply/to`.

Synchronous names can be used to support a functional programming style. A traditional example is the Fibonacci function.

```
# def fib(n) =
#   if n <= 1 then reply 1 to fib
#   else reply fib(n-1) + fib(n-2) to fib
# ;;
val fib : int -> int = <fun>
# print_int (fib 10)
# ;;
- : unit = ()
89
```

In contrast with Objective Caml `let` definitions, channel definitions are always potentially recursive.

Since synchronous channels have the same type and behave like functions, they seem useless. However there are significant differences, as explained by the next section on join patterns, and by the next chapter on distributed programming.

1.3 Join patterns

Join patterns significantly extend port name definitions.

1.3.1 Basics

A join pattern defines several ports simultaneously and specifies a synchronization pattern between these co-defined ports. For instance, the following source fragment defines two synchronizing port names `fruit` and `cake`:

```
# def fruit(f) & cake(c) = print_endline (f^" "^c) ; 0
# ;;
val fruit : string Join.chan = <abstr>
val cake : string Join.chan = <abstr>
```

To trigger the guarded process `print_endline (f^" "^c) ; 0`, messages must be sent on both `fruit` and `cake`.

```
# spawn fruit("apple") & cake("pie")
# ;;
- : unit = ()
apple pie
```

The parallel composition operator "`&`" appears both in join patterns and in processes. This highlights the kind of synchronization that the pattern matches.

Join definitions such as the one for `fruit` and `cake` provide a simple mean to express nondeterminism.

```
# spawn fruit "apple" & fruit "raspberry" & cake "pie" & cake "crumble"
# ;;
- : unit = ()
raspberry pie
apple crumble
```

Two cake names must appear on the console, but both combinations of fruits and cakes are correct.

Composite join definitions can specify several synchronization patterns.

```
# def apple() & pie() = print_string "apple pie" ; 0
#  or raspberry() & pie() = print_string "raspberry pie" ; 0
# ;;
val apple : unit Join.chan = <abstr>
val raspberry : unit Join.chan = <abstr>
val pie : unit Join.chan = <abstr>
```

Observe that the name `pie` is defined only once. Thus, `pie` potentially takes part in two synchronizations. This co-definition is expressed by the keyword `or`.

Again, internal choice is performed when only one invocation of `pie` is present:

```
# spawn apple() & raspberry() & pie()
# ;;
- : unit = ()
raspberry pie
```

1.3.2 ML pattern matching in join patterns

Up to now, we saw that the formal argument of a channel definition can be a variable or a tuple of variables. More generally, such a formal argument can be a pattern (in the Objective Caml pattern matching sense):

```
# type fruit = Apple | Raspberry | Cheese
# and desert = Pie | Cake
# ;;
type fruit = Apple | Raspberry | Cheese
and desert = Pie | Cake
# def f(Apple) & d(Pie) = echo_string("apple pie")
# or  f(Raspberry) & d(Pie) = echo_string("raspberry pie")
# or  f(Raspberry) & d(Cake) = echo_string("raspberry cake")
# or  f(Cheese) & d(Cake) = echo_string("cheese cake")
# ;;
val f : fruit Join.chan = <abstr>
val d : desert Join.chan = <abstr>
```

The definition above yields four competing behavior on the pair of channels `f` and `d`. For instance:

```
# spawn f(Raspberry) & d(Pie) & d(Cake)
# ;;
- : unit = ()
raspberry pie
```

And we get either "`raspberry pie`" or "`raspberry cake`".

The formal argument of channels can be any Objective Caml pattern. Here, by using "or-patterns" and `as` bindings for fruits, we can be more concise:

```
# let string_of_fruit = function
# | Apple -> "apple"
# | Raspberry -> "rasperry"
# | Cheese -> "cheese"
# ;;
val string_of_fruit : fruit -> string = <fun>
# def f(Apple|Raspberry as x) & d(Pie) = echo_string(string_of_fruit x^" pie")
# or  f(Raspberry|Cheese as x) & d(Cake) = echo_string(string_of_fruit x^" cake")
# ;;
val f : fruit Join.chan = <abstr>
val d : desert Join.chan = <abstr>
# spawn f(Raspberry) & d(Pie) & d(Cake)
# ;;
- : unit = ()
rasperry pie
```

And again the above display can be either desert with raspberry.

As another example, the following definition prints the sorted merge of two sorted lists sent as messages on channels `i1` and `i2`.

```
# def i1([]) & i2([]) = 0
# or  i1(x::xs) & i2([]) = print_int x ; i1(xs) & i2([])
# or  i1([]) & i2(y::ys) = print_int y ; i1([]) & i2(ys)
# or  i1(x::xs) & i2(y::ys) =
#       if x < y then begin print_int x ; i1(xs) & i2(y::ys) end
#       else if y < x then begin print_int y ; i1(x::xs) & i2(ys) end
#       else begin print_int x ; i1(xs) & i2(ys) end
# ;;
val i1 : int list Join.chan = <abstr>
val i2 : int list Join.chan = <abstr>

# spawn i1([1;3;4]) & i2([2;3])
# ;;
- : unit = ()
1234
```

It is important to notice that, by contrast with Objective Caml pattern matching, ambiguous matching are indeed ambiguous: as soon as a message matches a pattern it may be consumed, regardless of other receivers on the same channel.

```
# def c([]) = echo_string "Nil"
# or  c(_)  = echo_string "Anything"
# ;;
val c : 'a list Join.chan = <abstr>
# spawn c([])
# ;;
- : unit = ()
Anything
```

In the example above, you can see either "`Anything`" or "`Nil`" depending upon unspecified implementation details. To get the textual priority rule of Objective Caml matching semantics, use the `match` construct.

```
# def c(x) =
#   match x with
#   | [] -> echo_string "Nil"
#   | _  -> echo_string "Anything"
# ;;
val c : 'a list Join.chan = <abstr>
# spawn c([])
# ;;
- : unit = ()
Nil
```

1.3.3 Mixing asynchronous and synchronous channel definitions

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:

```
# def count(n) & inc() = count(n+1) & reply to inc
#  or count(n) & get() = count(n) & reply n to get
# ;;
val inc : unit -> unit = <fun>
val count : int Join.chan = <abstr>
val get : unit -> int = <fun>
# spawn count(0)
# ;;
- : unit = ()
```

This definition calls for two remarks. First, join pattern may mix synchronous and asynchronous message. Second, the usage of the name `count` above 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.

```
# spawn (inc() ; inc() ; 0) & (inc() ; 0)
# ;;
- : unit = ()
# def wait () =
#   let x = get () in
#   if x < 3 then wait ()
#   else begin
#     print_string "three is enough !!!" ; print_newline () ; 0
#   end
# ;;
val wait : unit Join.chan = <abstr>
# spawn wait ()
# ;;
- : unit = ()
three is enough !!!
```

Ensuring 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 will not work at all. This can be avoided by making the `count`, `inc` and `get` names local to a `create_counter` definition and then by exporting `inc` and `get` while hiding `count`, taking advantage of lexical scoping rules.

```
# let create_counter () =
#   def count(n) & inc0() = count(n+1) & reply to inc0
#    or count(n) & get0() = count(n) & reply n to get0 in
#   spawn count(0) ;
#   inc0, get0
# ;;
val create_counter : unit -> (unit -> unit) * (unit -> int) = <fun>
```

```
# let inc,get = create_counter ()
# ;;
val inc : unit -> unit = <fun>
val get : unit -> int = <fun>
```

This programming style is reminiscent of "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 `create_counter` creates a new object, initializes its internal state, and returns the exported methods. As a consequence, several counters may be allocated and used independently.

1.4 Control structures

Join pattern synchronization can express many common programming paradigms, either concurrent or sequential.

1.4.1 Some classical synchronization primitives

Locks

Join pattern synchronization can be used to emulate simple locks:

```
# let new_lock () =
#   def free() & lock() = reply to lock
#   and unlock() = free() & reply to unlock in
#   spawn free() ;
#   lock,unlock
# ;;
val new_lock : unit -> (unit -> unit) * (unit -> unit) = <fun>
```

Threads try to acquire the lock by performing a synchronous call on channel `lock`. Due to the definition of `lock()`, this consumes the name `free` and only one thread can get a response at a time. Another thread that attempts to acquire the lock is blocked until the thread that has the lock releases it by the synchronous call `unlock` that fires another invocation of `free`. As in Objective Caml, it is possible to introduce several bindings with the `and` keyword. These bindings are recursive.

To give an example of lock usage, we introduce a function that output its string argument several times:

```
# let print_n n s =
#   for i = 1 to n do
#     print_string s; Thread.delay 0.01
#   done
# ;;
val print_n : int -> string -> unit = <fun>
```

The `Thread.delay` calls prevents the same thread from running long enough to print all its strings.

Now consider two threads, one printing `*`'s, the other printing `+`'s.

```
# spawn (print_n 21 "*" ; 0) & (print_n 21 "+" ; 0)
# ;;
- : unit = ()
*+++++++++++++++++++++++++++++++++++++
```

As threads execute concurrently, their outputs may mix, depending upon scheduling. However, one can use a lock to delimit a critical section and prevent the interleaving of `*`'s and `+`'s.

```
# let lock, unlock = new_lock ()
# ;;
val lock : unit -> unit = <fun>
val unlock : unit -> unit = <fun>
# spawn begin
#     (lock() ; print_n 21 "*" ; unlock() ; 0)
#   & (lock() ; print_n 21 "+" ; unlock() ; 0)
# end
# ;;
- : unit = ()
*****************************************
```

Barriers

A barrier is another common synchronization mechanism. Basically, barriers define synchronization points in the execution of parallel tasks. Here is a simple barrier that synchronizes two threads:

```
# def join1 () & join2 () = reply to join1 & reply to join2
# ;;
val join1 : unit -> unit = <fun>
val join2 : unit -> unit = <fun>
```

The following two threads print `ba` or `ab` between matching parenthesis:

```
# spawn begin
#     (print_string "(" ; join1 () ; print_string "a" ; join1() ;
#      print_string ")" ; 0)
#   & (join2 () ; print_string "b" ; join2 () ; 0)
# end
# ;;
- : unit = ()
(ab)
```

Waiting for n events to occur

A frequent idiom is for a program to fork n concurrent tasks, and then to wait for those to complete. We can reformulate the informal "fork(s) and wait" as follows: channel `count` holds an integer that records the number of events still to be waited for. Events to be counted are materialized by sending empty messages on channel `tick`.

```
# def count(n) & tick() = count(n-1)
# or  count(0) & wait() = reply to wait
# ;;
val tick : unit Join.chan = <abstr>
val count : int Join.chan = <abstr>
val wait : unit -> unit = <fun>
```

By sending an initial message n on `count`, we enable counting n events.

```
# let n = 5
# ;;
val n : int = 5
# spawn count(n)
# ;;
- : unit = ()
```

Finally, one can wait for the n events to occur by calling the synchronous channel `wait`. The reaction rule `count(0) & wait() = reply to wait` is to be noticed: the formal argument of `count` is a *pattern*, here the integer constant `0`. This means that the guarded process will be firable only when message `0` is pending on channel `count` — See Section 1.3.2.

Beware, if at some time, `count(0)`, `tick()` and `wait()` are active, then the clause `count(n) & tick() = count(n-1)` can be selected. As an imediate consequence, `count(0)` will never be active again, and the agent waiting on `wait` will stay blocked. However, if at most n messages are sent on `tick`, then the device above works as expected.

The "count n events" idiom is so frequent that we build a countdown generator as follows:

```
# let create_countdown n =
#   def count(n) & tick() = count(n-1)
#   or  count(0) & wait() = reply to wait in
#   spawn count(n) ;
#   tick,wait
# ;;
val create_countdown : int -> unit Join.chan * (unit -> unit) = <fun>
```

Now we get a fresh countdown started at n by `create_countdown` n. More precisely, "a fresh countdown" is the pair of asynchronous `tick` and synchronous `wait`, where `wait` will answer once n messages are sent on `tick`. Also observe that the internal channel `count` is now kept secret and that the countdown generator takes care of initialisation.

In the following code, n messages are sent on `tick` by n asynchronous agents once they have printed one digit, while the closing parenthesis is printed only after `wait()` has returned.

```
# let n = 5
# ;;
val n : int = 5
# let tick,wait = create_countdown n
# ;;
val tick : unit Join.chan = <abstr>
val wait : unit -> unit = <fun>
```

```
# print_string "(" ;
# for i = 0 to n-1 do
#   spawn begin print_int i ; tick() end
# done ;
# wait () ;
# print_string ")"
# ;;
- : unit = ()
(02134)
```

As a result, the numbers `0`, `1`, `2`, `3`, `4` appear in any order, between parenthesis.

Collecting n results

A frequent variation on the idea of waiting for n events is collecting n results. Let us write slightly abstract code. We wish to combine n results using function f.

```
# let create_collector f y0 n =
#   def count(y,n) & collect(x) = count(f x y,n-1)
#   or  count(y,0) & wait() = reply y to wait in
#   spawn count(y0,n) ;
#   collect, wait
# ;;
val create_collector :
  ('a -> 'b -> 'b) -> 'b -> int -> 'a Join.chan * (unit -> 'b) = <fun>
```

The `create_collector` function is quite similar to the `create_countdown` function. Additionnaly, it accumulate messages sent to `collect` (`tick` in the case of countdown) into the first component of the internal message on `count`. Incidentally, observe that `create_countdown` can be implemented by using the more general `create_collector`.

```
# let create_countdown n = create_collector (fun () () -> ()) () n
# ;;
val create_countdown : int -> unit Join.chan * (unit -> unit) = <fun>
```

The following `collect_as_sum` will collect n integers on channel `add` and send their sum as the result of synchronous `wait`.

```
# let collect_as_sum n =
#   let add, wait = create_collector (+) 0 n in
#   add,wait
# ;;
val collect_as_sum : int -> int Join.chan * (unit -> int) = <fun>
```

One now easily computes the sum of the n first integers as follows:

```
# let n = 10
# ;;
val n : int = 10
```

```
# let add,wait = collect_as_sum n
# ;;
val add : int Join.chan = <abstr>
val wait : unit -> int = <fun>
# spawn for i=0 to n-1 do add(i) done
# ;;
- : unit = ()
# print_int (wait())
# ;;
- : unit = ()
45
```

It is to be noticed that collectors are provided by the JoCaml standard library module `JoinCount.Collector`.

```
# module Col = JoinCount.Collector
# ;;
module Col :
  sig
    type ('a, 'b) t =
      ('a, 'b) JoinCount.Collector.t = {
      collect : 'a Join.chan;
      wait : unit -> 'b;
    }
    val create : ('a -> 'b -> 'b) -> 'b -> int -> ('a, 'b) t
  end
```

Generally speaking, the standard library of JoCaml makes heavy use of records, as illustrated by the type of collectors (`JoinCount.Collector.t` above). As a matter of fact, records offer a convenient way to pack together the public channels of join-definitions.

Here is another way of summing the first `n` integers asynchronouosly, that time relying on the collectors of the standard library.

```
# let col = Col.create (+) 0 n
# ;;
val col : (int, int) Col.t = {Col.collect = <abstr>; Col.wait = <fun>}
# spawn for i=0 to  n-1 do col.Col.collect(i) done
# ;;
- : unit = ()
# print_int (col.Col.wait())
# ;;
- : unit = ()
45
```

In the code above, one should probably noticed that the field names of records are given in fully qualified notation as, for instance, `col.Col.wait()`.

Bi-directional channels

Bi-directional channels appear in most process calculi. 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. In contrast, A JoCaml process that knows a channel can only send messages on it, whereas a unique channel definition receives all messages. Finally, the scope of a pi-calculus channel name c is defined by the `new` c `in` P operator. Such an operator does not exist in JoCaml, since join definitions are binding constructs.

Nonetheless, bi-directional channels can be defined in JoCaml as follows:

```
# let new_pi_channel () =
#   def send(x) & receive() = reply x to receive in
#   send, receive
# ;;
val new_pi_channel : unit -> 'a Join.chan * (unit -> 'a) = <fun>
```

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`. Let us now "translate" the pi-calculus process

```
new c,d in c![1] | c![2] | c?x.d![x+x] | d?y. print(y))
```

We get:

```
# spawn begin
#   let sc, rc = new_pi_channel ()
#   and sd, rd = new_pi_channel () in
#     sc(1) &  sc(2) &  (let x = rc() in sd(x+x))
#   & (let y = rd() in print_int y ; 0)
# end
# ;;
- : unit = ()
4
```

Observe that, by contrast with the JoCaml semantics of receptors, the process that performs `x+x` runs only once.

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

```
# let new_pi_sync_channel () =
#   def send(x) & receive() = reply x to receive & reply to send in
#   send, receive
# ;;
val new_pi_sync_channel : unit -> ('a -> unit) * (unit -> 'a) = <fun>
```

1.4.2 Timeout

We can define a function `timeout` which tries to compute a function `f`. If the computation of `f` is too long, we do not wait for the result. Instead, we acknowledge that a timeout occurred, by returning value `None`.

```
# let timeout t f x =
#   def wait() & finished(r) = reply Some r to wait
#   or  wait() & timeout() = reply None to wait in
#   spawn begin
#     finished(f x) &
#     begin Thread.delay t; timeout() end
#   end ;
#   wait()
# ;;
val timeout : float -> ('a -> 'b) -> 'a -> 'b option = <fun>
```

It is to be noticed that, in case a timeout occurs, the computation of `f` is not interrupted. Namely, there is no way to "kill" a running agent. However if computing `f x` takes more time than the timeout value, `timeout f x` will return `None`. It should perhaps be observed that we here rely upon the underlying (Objective Caml) thread implementation, *i.e.* we use the function `Thread.delay` from the `Thread` library.

As an example of a computation that takes a long time to execute, we select an infinite loop.

```
# let rec loop () =  Thread.yield () ; loop ()
# ;;
val loop : unit -> 'a = <fun>
```

Observe that we call `Thread.yield ()`, so as to give other threads a chance to be scheduled. Here again we rely upon the underlying thread scheduler.

Finally, we execute `loop ()` under the control of a timeout of `0.5` second.

```
# match timeout 0.5 loop () with
# | None -> print_string "Timeout !"
# | Some _ -> print_string "No timeout ! "
# ;;
- : unit = ()
Timeout !
```

Timeout occurred in the sense that the call to the `timeout` function returns after about half a second. However, the controlled computation is still alive, consuming resources. For solving this issue, see Section 1.10.1.

1.4.3 Iterations

Join patterns are also useful for expressing various programming control structures. Our first example deal with iterations on an integer interval.

Simple loop

Asynchronous loops can be used when the execution order for the iterated actions is irrelevant.

```
# def loop(a,x) = if x > 0 then (a() & loop(a,x-1))
# ;;
val loop : (unit Join.chan * int) Join.chan = <abstr>
```

```
# def echo_star() = print_string "*" ; 0
# ;;
val echo_star : unit Join.chan = <abstr>
# spawn loop(echo_star,5)
# ;;
- : unit = ()
*****
```

There is also a `for` loop at the process level.

```
# def loop(a,x) = for i=1 to x do a() done
# ;;
val loop : (unit Join.chan * int) Join.chan = <abstr>
# spawn loop(echo_star,5)
# ;;
- : unit = ()
*****
```

List iterator

In this section we examine the asynchronous counterpart of the list iterator `List.iter`:

```
# let iter f xs = List.iter (fun x -> spawn f(x)) xs
# ;;
val iter : 'a Join.chan -> 'a list -> unit = <fun>
```

Hence `iter` c $[e_1;e_2;\ldots;e_n]$ will act as $c(e_1)\&c(e_2)\&\cdots\&c(e_n)$.

```
# let digits = [0;1;2;3;4;5;6;7;8;9]
# ;;
val digits : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
# iter echo digits
# ;;
- : unit = ()
0213897654
```

Of course, `iter` can be written as a channel and without using `List.iter`.

```
# let iter c =
#   def do_iter(x::xs) = c(x) & do_iter(xs)
#   or  do_iter([]) = 0 in
#   (fun xs -> spawn do_iter(xs))
# ;;
val iter : 'a Join.chan -> 'a list -> unit = <fun>
# iter echo digits
# ;;
- : unit = ()
0213548967
```

For the iteration over a list to produce a result (*i.e.* to get the asynchronous counterpart of `List.fold`), one combines the asynchronous iterator and the collector of Section 1.4.1. For instance, here is how one can sum the squares of all the integers in a list.

```
# let square x = x*x
# ;;
val square : int -> int = <fun>
# let sum xs =
#   let add,wait = collect_as_sum (List.length xs) in
#   def add_square(x) = add(square x) in
#   iter add_square xs ;
#   wait()
# ;;
val sum : int list -> int = <fun>
# print_int (sum digits)
# ;;
- : unit = ()
285
```

Another variation on the same idea is making a list of squares, order being irrelevant.

```
# let squares xs =
#   let add,wait = create_collector (fun x xs -> x::xs) [] (List.length xs) in
#   def add_square(x) = add(square(x)) in
#   iter add_square xs ;
#   wait()
# ;;
val squares : int list -> int list = <fun>
# squares digits
# ;;
- : int list = [81; 49; 64; 25; 36; 16; 0; 1; 9; 4]
```

As such, the functions `sum` and `squares` are not very interesting, however they may act as an introduction to the more involved topic of distributed iteration.

Distributed iterations

Sharing iterations between several "agents" requires more work. Let us informally define an agent as some computing unit. In this section, an agent is represented by a synchronous channel. In a more realistic setting, different agents would reside on different computers.

For instance, here are two agents `square1` and `square2`. The agent `square1` models a fast machine, whereas `square2` models a slow machine thanks to an artificial delay.

```
# let square1 i =
#   print_string "(" ;
#   let r = i*i in
#   print_string ")" ;
#   r
```

```
# ;;
val square1 : int -> int = <fun>
# let square2 i =
#   print_string "<" ;
#   let r = i*i in
#   Thread.delay 0.5 ;
#   print_string ">" ;
#   r
# ;;
val square2 : int -> int = <fun>
```

Additionally, `square1` and `square2` differ marginally by their console output: the fast `square1` prints "(" when it starts and ")"' when it's done, while `square2` uses "<" and ">" for the same purpose.

Sharing a loop between several agents is allocating the iterations to be performed to them. The following channel `create_sum`, returns a `register` channel and a `wait` channel. An agent registers by sending its computing channel on `register`. The final loop result is returned on `wait`.

```
# let create_sum n =
#   let add,wait = collect_as_sum n in
#   def add_square(x) & register(square) = add(square(x)) & register(square) in
#   for i=0 to n-1 do spawn add_square(i) done ;
#   register, wait
# ;;
val create_sum : int -> (int -> int) Join.chan * (unit -> int) = <fun>
```

The key difference with the asynchronous `sum` loop from the previous section resides in an indirection: instead of sending `x` squared directly on the channel `add`, the new `add_square` channel extracts a squaring function from the channel `register`. As a consequence, the agents `square1` and `square2` may now compete for loop iterations, provided two invocations `register(square1)` and `register(square2)` are active.

```
# let register, wait = create_sum 32
# ;;
val register : (int -> int) Join.chan = <abstr>
val wait : unit -> int = <fun>
# spawn register(square2) & register(square1)
# ;;
- : unit = ()
# print_int (wait ());
# ;;
- : unit = ()
<<()<<()<()()<<()()<<()()<()<()()<<()()<()()()<<>>>>>>>>>>>>>>>>>10416
```

The distributed loop above is not satisfactory, since it does not take the relative computing speed of `square1` and `square2` into account while allocating iterations. The `add(square(x))` jobs are spawned asynchronously, so that all iterations are performed concurrently. As a result, the iteration space is partitioned randomly between `square1` and `square2`, as illustrated by the output above, which can be about anything. This leads to a poor load balance, in fact to no load balance at all.

A better solution is for an agent to execute its share of work in sequence rather than concurrently. This is achieved by the slightly modified definition for `def add_square(x) & register(square)` below:

```
# let create_sum n =
#    let add,wait = collect_as_sum n in
#    def add_square(x) & register(square) =
#      let r = square x in add(r) & register(square) in
#    for i=0 to n-1 do spawn add_square(i) done ;
#    register, wait
# ;;
val create_sum : int -> (int -> int) Join.chan * (unit -> int) = <fun>
```

In the new definition, `register(square)` is launched again only once `square(x)` is computed. This is so because of the semantics of `let` $x = E$ `in` P, which says that process P executes only once expression E is evaluated.

```
# let register, wait = create_sum 32
# ;;
val create_sum : int -> (int -> int) Join.chan * (unit -> int) = <fun>
val register : (int -> int) Join.chan = <abstr>
val wait : unit -> int = <fun>
# spawn register(square1) & register(square2)
# ;;
- : unit = ()
# print_int (wait ())
# ;;
- : unit = ()
()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()10416
```

Finally, at the fine tuning level, one may object that messages on the `add_square` channel are a bit out of control: we send a burst of such message with a `for` loop, and they probably accumulate in some internal queue, waiting for agents to retrieve them.

A final version of `create_sum` takes care not to send too many messages on `add_square` concurrently. Basically, we only have to send a new message as soon as a registred agent has retrieved one message.

```
# let create_sum n =
#    let add,wait = collect_as_sum n in
#    def add_square(x) & register(square) =
#      (if x > 0 then add_square(x-1)) &
#      (let r = square x in register(square) & add(r)) in
#    spawn add_square(n-1) ;
#    register, wait
# ;;
val create_sum : int -> (int -> int) Join.chan * (unit -> int) = <fun>
```

```
# let register, wait = create_sum 32
# ;;
val register : (int -> int) Join.chan = <abstr>
val wait : unit -> int = <fun>
# spawn register(square1) & register(square2)
# ;;
- : unit = ()
# print_int (wait ())
# ;;
- : unit = ()
()()()()()<()()()()()()()()()()()()()()()()()()()()()()()()()()()()()>10416
```

1.5 Data structures

1.5.1 A concurrent reference cell

Object states, represented as join patterns, can be altered by invoking the appropriate methods. Here is a definition for a reference cell. One method (`get`) examines the content of the cell, while another (`put`) alters it.

```
# let create_ref y0 =
#   def state(y) & get() = state(y) & reply y to get
#    or state(_) & put(y) = state(y) & reply to put in
#   spawn state(y0) ;
#   (get, put)
# ;;
val create_ref : 'a -> (unit -> 'a) * ('a -> unit) = <fun>
```

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

```
# let gi, pi = create_ref 0
# and gs, ps = create_ref ""
# ;;
val gi : unit -> int = <fun>
val pi : int -> unit = <fun>
val gs : unit -> string = <fun>
val ps : string -> unit = <fun>
```

1.5.2 A concurrent stack

A stack is a data structure that provide `push` and `pop` operations with LIFO (Last In First Out) semantics.

```
# let new_stack () =
#   def state (s) & push (v) = state (v::s) & reply to push
#    or state (x::s) & pop () = state (s) & reply x to pop in
```

```
#   spawn state([]);
#   pop, push
# ;;
val new_stack : unit -> (unit -> 'a) * ('a -> unit) = <fun>
```

The first join pattern (`state(s) & push(v)`) is an ordinary one: it is matched whenever there are messages on both `state` and `push`. The second join pattern (`state (x::s) & pop ()`) uses algebraic pattern matching. This pattern is matched only when there are messages on both `state` and `pop` and the `state` content is an non-empty list. As a consequence, an attempt to retrieve an element from an empty stack is not an error: answering to `pop` is simply postponed until the stack fills in.

```
# let pop, push = new_stack ()
# ;;
val pop : unit -> '_a = <fun>
val push : '_a -> unit = <fun>
# spawn echo(pop())
# ;;
- : unit = ()
# push(1)
# ;;
- : unit = ()
1
```

1.5.3 Buffers

A buffer is some kind of double-ended queue. Elements are added at the one end with `put` and retrieved at the other end with `get`. That is, a buffer preserves elements ordering.

```
# type 'a buffer = { put : 'a -> unit ; get : unit -> 'a ; }
# ;;
type 'a buffer = { put : 'a -> unit; get : unit -> 'a; }
```

For us, `put` and `get` are synchronous channels, and a `get` attempt on an empty buffer is blocking. By contrast a `put` attempt always succeeds: our buffer is unbounded.

Using the trick of encoding a FIFO queue functionnaly as a pair of lists, we write:

```
# let create_buffer () =
#   def alive (xs,y::ys) & get() = alive(xs,ys) & reply y to get
#   or alive(_::_ as xs,[]) & get() = alive([], List.rev xs) & reply get() to get
#   or alive(xs,ys) & put(x) = alive(x::xs,ys) & reply to put in
#   spawn alive([],[]) ;
#   {put=put; get=get;}
# ;;
val create_buffer : unit -> 'a buffer = <fun>
```

We shall assume that the buffer is used by two communicating agents: one *writer* that performs `put` and one *reader* that performs `get`. The buffer then posseses the property that the reader reads the

elements in the same order as the writer wrote them. More precisely, the channel `alive(xs,ys)` is used in internally to encode the state of the buffer: the ordered list of elements seen from reader's side (`get`) being `ys@List.rev xs`. One may observe that calls on `get` are blocked when the buffer is empty, *i.e.* when `alive([],[])` is active.

1.6 A serious example: connecting a producer with consumers

To demonstrate that our buffer is useful we consider a simple producer–consumer(s) problem, our producer feeds our consumer with `1,2,3,4,5`

```
# type 'a consumer = { send : 'a -> unit ; }
# ;;
type 'a consumer = { send : 'a -> unit; }
# let producer c = for k = 1 to 5 do c.send k done
# ;;
val producer : int consumer -> unit = <fun>
```

And here is a simple consumer that prints its input on the console.

```
# let consumer = {send=print_int}
# ;;
val consumer : int consumer = {send = <fun>}
```

We connect the producer and the consumer by the means of ordinary function application.

```
# producer consumer
# ;;
- : unit = ()
12345
```

We introduced the record type `'a consumer` for getting the compiler to issue significant types. However, a consumer is nothing more than a function of type `'a -> unit`. Thus, we have just explained function application in a rather contrieved manner.

Now we start complicating things: we want to feed two consumers. Those consumers are "concurrent agents" and we encode them as join definitions.

```
# let fast_consumer =
#   let fast_cons x xs =  Thread.delay 0.1 ; x::xs in
#   def put(x) & state(xs) =
#     print_string ("("^string_of_int x) ;
#     let xs = fast_cons x xs in
#     print_string ")" ;
#     state(xs) & reply to put in
#  spawn state([]) ;
#  { send=put }
# and slow_consumer =
#   let slow_cons x xs = Thread.delay 0.3 ; x::xs in
#   def put(x) & state(xs) =
```

```
#     print_string ("<"^string_of_int x) ;
#     let xs = slow_cons x xs in
#     print_string ">" ;
#     state(xs) & reply to put in
#   spawn state([]) ;
#   { send=put; }
# ;;
val fast_consumer : int consumer = {send = <fun>}
val slow_consumer : int consumer = {send = <fun>}
```

Both consumers accept integers on the synchronous channel `put` and accumulate integers as a list in some internal channel `state`. However, the fast consumer builds lists at about three times the speed of the slow consumer. We assume that element order is critical: consumers must compute the reversed list of producer output. Consumers also differ marginally as regards console output, the fast consumer prints recieved value x as (x), while the slow consumer prints $\texttt{<}x\texttt{>}$.

We do not intend the consumers to compete for producer output. Instead, they should get their own copy. Of course, we wish not change the producer. Hence, we insert the following duplicator between the producer and the consumers.

```
# let dup c1 c2 =
#   let dup_send x = c1.send x ; c2.send x in
#   { send=dup_send }
# ;;
val dup : 'a consumer -> 'a consumer -> 'a consumer = <fun>
```

Clearly, given the type above, the producer should not notice that its messages are duplicated: the combined consumers act as single consumer.

We marginally alter the producer, so as it shows timing information.

```
# let producer c =
#   let t_start = Unix.gettimeofday () in
#   for k = 1 to 5 do c.send k done ;
#   let t_end = Unix.gettimeofday () in
#   print_string (Printf.sprintf "{Time elapsed: %0.2f}" (t_end -. t_start))
# ;;
val producer : int consumer -> unit = <fun>
```

And then we connect the producer and both consumers.

```
# producer (dup slow_consumer fast_consumer)
# ;;
- : unit = ()
<1>(1)<2>(2)<3>(3)<4>(4)<5>(5){Time elapsed: 2.00}
```

Integers are consummed (and printed) in the right order. However there is no concurrency at all: consumers execute sequentially and the producer waits for both consumers.

A first idea to have consumers to execute concurrentlty is to send the integers to them asynchronously. Ideally, we should change the type of consumers making it to be `{send : 'a Join.chan}`. So as not to rewrite everything, we write a wrapper that simply spawns calls to the `send` function of consumers.

```
# let asynchronyze c = { send=(fun x -> spawn (c.send(x); 0)) }
# ;;
val asynchronyze : 'a consumer -> 'a consumer = <fun>
# let async_slow = asynchronyze slow_consumer
# and async_fast = asynchronyze fast_consumer
# ;;
val async_slow : int consumer = {send = <fun>}
val async_fast : int consumer = {send = <fun>}
# producer (dup async_slow async_fast)
# ;;
- : unit = ()
<1(1{Time elapsed: 0.01})(5)(4><5)(3)(2)><4><3><2>
```

Now consumers execute concurrently and producer speed is unconstrained. From the point of view of the producer, we are connected to a fast consumer. From the point of view of a consumer, we select integers at random, amongst a soup of asynchronous messages. Consumers can execute concurrently, because they are messages for both of them in the soup simultaneously. However, as a result of all messages being sent asynchronously, the ordering on producer output is lost.

To recover that ordering while preserving concurrent execution, we can use buffers.

```
# let bufferize c =
#   let buff = create_buffer () in
#   def transmit() = c.send(buff.get()) ; transmit() in
#   spawn transmit() ;
#   { send=buff.put }
# ;;
val bufferize : 'a consumer -> 'a consumer = <fun>
```

The `bufferize` function takes a consumer as its argument and return another consumer. The returned consumer simply puts messages into an internal buffer `buff`. A concurrent agent `transmit` extracts the messages from the buffer and forwards them to the initial consumer `c`.

The bufferizing duplicator inserts a buffer in front of each consumer.

```
# let dup_buffered c1 c2 = dup (bufferize c1) (bufferize c2)
# ;;
val dup_buffered : 'a consumer -> 'a consumer -> 'a consumer = <fun>
# producer (dup_buffered slow_consumer fast_consumer)
# ;;
- : unit = ()
(1<1{Time elapsed: 0.01})(2)(3><2)(4)(5)><3><4><5>
```

From the point of view of the producer, a buffer is a fast consumer, hence the producer still runs at full speed. But now, between the producer and a consumer, there is a buffer that preserves ordering, in place of a soup that mixes everything. Additionally, the consumer is now a concurrent agent that attempt to get a new value from the buffer, as soon as it is done with the previous value. As a result, consumers execute concurrently, and consume producer output in the issuing order.

We may feel satisfied, but we are not done yet: we may now consider that the producer is too fast and that it fills the buffers needlessly. It may be a good idea for the producer to wait for one

message to be consumed by someone, before issuing the next message. That way, we avoid filling up buffers. To that aim, we attach a fresh `tick` channel to each message issued by the producer, and then wait for a message on `tick`.

```
# let add_tick c =
#   let send_ticked x =
#     def wait() & tick() = reply to wait in
#     c.send (tick,x) ;
#     wait () in
#   { send=send_ticked }
# ;;
val add_tick : (unit Join.chan * 'a) consumer -> 'a consumer = <fun>
# let ticked_producer c = producer (add_tick c)
# ;;
val ticked_producer : (unit Join.chan * int) consumer -> unit = <fun>
```

At the consuming end, we remove the `tick`, transmit the actual message to the actual consumer, and then, once the messsage is accepted, issue a message on `tick`.

```
# let remove_tick c =
#   let send_unticked (tick,x) = c.send x ; spawn tick () in
#   { send=send_unticked }
# ;;
val remove_tick : 'a consumer -> (unit Join.chan * 'a) consumer = <fun>
# let ticked_slow = remove_tick slow_consumer
# and ticked_fast = remove_tick fast_consumer
# ;;
val ticked_slow : (unit Join.chan * int) consumer = {send = <fun>}
val ticked_fast : (unit Join.chan * int) consumer = {send = <fun>}
```

Then, we combine the ticked producer and the ticked consumers as we did for unticked ones.

```
# ticked_producer (dup_buffered ticked_slow ticked_fast)
# ;;
- : unit = ()
<1(1)(2)(3><2)(4)(5){Time elapsed: 0.52}><3><4><5>
```

As we can see, producer speed is now more or less the speed of the fast consumer.

1.7 Modules

JoCaml relies on the same module system as Objective Caml. For example, a `Semaphore` module can be defined with the following interface.

```
# module type Semaphore =
#   sig
#     type t
#     val create: int -> t
```

```
#     val p: t -> unit
#     val v: t -> unit
#   end
# ;;
module type Semaphore =
  sig type t val create : int -> t val p : t -> unit val v : t -> unit end
```

The type `t` of a semaphore is an abstract type. Function `create n` returns a new semaphore initialized with the value `n`. `p` and `v` are the atomic functions to manipulate semaphores.

The implementation of semaphores can be done as the one of locks (see section 1.4.1).

```
# module Semaphore: Semaphore =
#   struct
#     type t = (unit -> unit) * (unit -> unit)
#     let create n =
#       def p() & s() = reply to p
#       and v() = s() & reply to v in
#       for i = 1 to n do spawn s() done;
#       (p,v)
#     let p (prolagen,_) = prolagen()
#     and v (_,verhogen) = verhogen()
#   end
# ;;
module Semaphore : Semaphore
```

And here an example of the usage of module `Semaphore`.

```
# let s = Semaphore.create 2
# ;;
val s : Semaphore.t = <abstr>
# def agent(x) =
#   def loop(n,k) = match n with
#     | 0 -> k()
#     | n -> print_int x ; loop(n-1,k) in
#   Semaphore.p s ;
#   loop(5,def k() = Semaphore.v s ; 0 in k)
# ;;
val agent : int Join.chan = <abstr>
# spawn agent(1) & agent(2) & agent(3)
# ;;
- : unit = ()
233333222211111
```

As to semaphore semaphore semantics, in the ouput above there must be an “initial value” (maybe `1`) and a “final value” (maybe `3`) such all instances of the initial value apear before the instances of the final value.

1.8 A word on typing

The JoCaml type system is derived from the ML type system and it should be no surprise to ML programmers. The key point in typing à la ML is *parametric* polymorphism. For instance, here is a polymorphic identity function:

```
# def id(x) = reply x to id
# ;;
val id : 'a -> 'a = <fun>
```

The type for `id` contains a type variable "`'a`" that can be instantiated to any type each time `id` is actually used. Such a type variable is a generalized type variable. For instance, in the following program, variable "`'a`" is instantiated successively to `int` and `string`:

```
# let i = id(1) and s = id("coucou")
# ;;
val i : int = 1
val s : string = "coucou"
# print_int i ; print_string s
# ;;
- : unit = ()
1coucou
```

In other words, the first occurrence of `id` above has type `int -> int`, while the second has type `string -> string`. Experienced ML programmers may wonder how 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:

```
# def state(x) & get() = state(x) & reply x to get
# or  state(_) & set(x) = state(x) & reply to set
# ;;
val get : unit -> '_a = <fun>
val state : '_a Join.chan = <abstr>
val set : '_a -> unit = <fun>
```

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

More specifically, if "`'_a`" is instantiated to some type `int`, by sending the message `0` on `state`. Then, the type for `get` is `unit -> int` in the rest of the program, as shown by the type for `x` below. As a consequence, the following program does not type-check and a runtime type-error (printing an integer, while believing it is a string) is avoided:

```
# def state(x) & get() = state(x) & reply x to get
# or  state(_) & set(x) = state(x) & reply to set
# ;;
```

```
val get : unit -> '_a = <fun>
val state : '_a Join.chan = <abstr>
val set : '_a -> unit = <fun>
# spawn state(0)
# ;;
- : unit = ()
# let x = get ()
# ;;
val x : int = 0
# print_string x
# ;;
Error: This expression has type int but is here used with type string
```

Non generalized type variables appear when the type of several co-defined port names share a type variable. Such a type variable is not generalized.

```
# def port(p) & arg(x) = p x
# ;;
val port : '_a Join.chan Join.chan = <abstr>
val arg : '_a Join.chan = <abstr>
```

A workaround is to encapsulate the faulty names into a function definition. This restores polymorphism.

```
# let create_it () = def port(p) & arg(x) = p x in  port,arg
# ;;
val create_it : unit -> 'a Join.chan Join.chan * 'a Join.chan = <fun>
```

Non-generalized type variables also appear in the types of the identifiers defined by a value binding.

```
# let (p1,a1),(p2,a2) = create_it (), create_it ()
# ;;
val p1 : '_a Join.chan Join.chan = <abstr>
val a1 : '_a Join.chan = <abstr>
val p2 : '_a Join.chan Join.chan = <abstr>
val a2 : '_a Join.chan = <abstr>
# spawn p1(echo) & p2(echo_string)
# ;;
- : unit = ()
# (a1,a2)
# ;;
- : int Join.chan * string Join.chan = (<abstr>, <abstr>)
# spawn a1(1) & a2("coucou")
# ;;
- : unit = ()
coucou1
```

It is interesting to notice that invoking `create_it ()` twice yields two different sets of `port` and `arg` port names, whose types contain different type variables — unfortunately all type variables

appear as `'_a`. Namely, once the variables are instantiated by sending messages on `p1` and `p2`, the types of `a1` and `a2` are instantianted accordingly. Thereby, programmers make explicit the different type instantiations that are performed silently by the compiler in the case of generalized type variables.

1.9 Exceptions

Since processes are mapped to several threads at run-time, it is important to specify their behaviors in the presence of exceptions.

Exceptions behave as in Objective Caml for Objective Caml expressions. If the exception is not caught in the expression, the behavior will depend on the way the process as been spawned. In the following, processes that must reply to a synchronous channel are called synchronous processes and the others are asynchronous processes.

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

```
# spawn begin
#     (failwith "Bye bye"; 0)
#   & (for i = 1 to 10 do print_int i done; 0)
# end
# ;;
Thread 3 killed on uncaught exception Failure("Bye bye")
- : unit = ()
12345678910
```

To avoid the error message, one can raise the `Join.Exit` exception. Then, the process that commits suicide does so silently.

```
# spawn begin
#     (raise Join.Exit ; 0)
#   & (for i = 1 to 10 do print_int i done; 0)
# end
# ;;
- : unit = ()
12345678910
```

An exception raised in a process that includes the `reply` construct behaves differently: the process waiting for the result will receive the exception, which will be propagated as in an Objective Caml function.

```
# def die() = failwith "die"; reply to die
# ;;
val die : unit -> unit = <fun>
# try
#   die()
# with
#   Failure msg -> print_string (Printf.sprintf "dead on '%s'\n" msg)
```

```
# ;;
- : unit = ()
dead on 'die'
```

Synchronous processes may be in charge of replying to more than one synchronous call at the time when an exception is raised. In other words there can be several reply constructs that are syntactically guarded by a shared expression that raises the exception. In such cases, the exception is duplicated and thrown at all threads, reversing joins into forks.

```
# def a() & b() = failwith "die"; reply to a & reply to b
# ;;
val a : unit -> unit = <fun>
val b : unit -> unit = <fun>
# spawn begin
#     ( (try a() with Failure _ -> print_string "a failed\n"); 0 )
#   & ( (try b() with Failure _ -> print_string "b failed\n"); 0 )
# end
# ;;
- : unit = ()
a failed
b failed
```

Transmission of exceptions by reply constructs follow relatively straightforward rules. Let us first define a channel `protect` to control our experiments.

```
# def protect(name,c) =
#   begin try c() ; print_string (name ^ ": success\n")
#   with Failure _ ->  print_string (name ^ ": failure\n") end ;
#   0
# ;;
val protect : (string * (unit -> 'a)) Join.chan = <abstr>
```

Then, the rule is as follows: basically, the exception is transmitted if the reply is to be executed after the exception, following standard evaluation rules. For instance, in "E ; P", expression E evaluates before process P executes, while in "P_1 & P_2" processes P_1 and P_2 execute independantly.

```
# def a() & b() = (failwith "die"; reply to a) & reply to b
# ;;
val a : unit -> unit = <fun>
val b : unit -> unit = <fun>
# spawn protect("a",a) & protect("b",b)
# ;;
- : unit = ()
a: failure
b: success
```

And here is another test, with three channels.

```
# def a() & b() & c() =
#   reply to c &
```

```
#   if failwith "die" then reply to a & reply to b
#   else reply to b & reply to a
# ;;
val a : unit -> unit = <fun>
val b : unit -> unit = <fun>
val c : unit -> unit = <fun>
# spawn protect("a",a) & protect("b",b) & protect("c",c)
# ;;
- : unit = ()
c: success
a: failure
b: failure
```

1.10 A complete example: controlling several remote shell executions

1.10.1 Realistic timeouts

In section 1.4.2, we introduced the basics of programming a timeout in JoCaml. A first step in a more realistic timeout, is for the controlled computation to abort when the timeout expires. To that aim, the controlled computation must give a means to kill itself. Here, we define a infinite loop that computes a function `step` at each iteration and that can be killed between two iterations.

```
# exception Killed
# ;;
exception Killed
# let create_loop (step) =
#   def run() & ok() = ok() & reply step(); run() to run
#   or  run() & killed() = reply raise Killed to run
#   or  kill() & ok() = killed() in
#   let loop () = spawn ok() ; run () in
#   loop,kill
# ;;
val create_loop : (unit -> 'a) -> (unit -> 'b) * unit Join.chan = <fun>
# let run,kill = create_loop (fun () -> print_string "*"; Thread.delay 0.01)
# ;;
val run : unit -> 'a = <fun>
val kill : unit Join.chan = <abstr>
```

The channels `ok` and `killed` are used internally, they express the status of the computing agent. At a given time, there is a message pending on either `ok` or `killed`. In the first situation, computation can go on; while in the second situation, computation is interrupted, and an exception is raised, as a reply to whoever called `run`. The computation is controlled from outside by the means of one function `loop` to compute a result (here to loop), and of one asynchronous channel `kill` to stop computing.

The new `timeout` function is a small improvement over the previous one: when the delay has expired we also kill the computation by sending a message on the adequate channel `kill`, which is passed to `timeout` for that purpose.

```
# let timeout t f x kill =
#   def wait () & finished (r) = reply Some r to wait
#   or wait () & timeout () = kill () & reply None to wait
#   in
#   spawn begin
#     finished (f x) &
#     begin Thread.delay t ; timeout() end
#   end ;
#   wait()
# ;;
val timeout : float -> ('a -> 'b) -> 'a -> unit Join.chan -> 'b option =
  <fun>
# match timeout 0.5 run () kill with
# | None -> print_string "Timeout! "
# | Some _ -> print_string "No timeout! "
# ;;
- : unit = ()
Thread 10 killed on uncaught exception Killed
*****************************************Timeout!
```

The message `Thread X killed on uncaught exception Killed` above is issued by the JoCaml system, as an indication that some of the underlying threads terminated on an abnormal condition. Here, the killed thread is in charge of executing the process `finished (f x)`. However, the evaluation of `f x` does not result in a value to be sent on channel `finished` — instead, exception `Killed` is raised, and the thread in charge terminates abnormally, since there is no value to send.

To get rid of the message one may use the `Join.Exit` exception (see Section 1.9). However, it may be advisable not to use system exception in place of users exception. Thus, one can also replace the simple process `finished (f x)` by the more complex one:

```
let r = try Some (f x) with Killed -> None in
match r with Some r -> finished(r) |  None -> 0
```

1.10.2 Forking an Unix process under timeout control

The Unix command `arch` echoes a conventional string that describes the architecture of the machine, such as `alpha`, `i686` etc. The Unix command `rsh` *host cmd* performs the remote execution of command *cmd* on *host*. In particular, `rsh` copies the standard output of the remote command to its standard output. Hence by issuing the command `rsh` *host* `arch`, we get the architecture of *host*. By a using a bit of classical Unix programming the following function forks an Unix process that performs command `arch` on a remote host.

```
# let rsh_arch host = JoinProc.open_in "/usr/bin/rsh" [| host ; "arch" |]
# ;;
val rsh_arch : string -> int * in_channel = <fun>
```

The `rsh_arch` function is in charge of forking the concurrent Unix process that performs the remote `arch` command. To that end it calls `open_in` from the standard library module `JoinProc`. Observe that the function `rsh_arch` returns a process id and an input (IO) channel. Thanks to the Unix plumbing that `JoinProc.open_in` performs, the output of the remote `arch` command can be read on this channel. Moreover, the process id is the information needed to kill an Unix process, as we should do when timeout expires.

Now, to use the `timeout` function of the previous section, we disguise the remote call to `arch` into a pair of a "run" function and of a "kill" channel.

```
# let create_arch host =
#   let pid, inchan = rsh_arch host in
#   def result(r) & wait() & ok () = reply r to wait
#   or kill() & ok() =
#      begin try Unix.kill pid Sys.sigkill with _ -> () end ; 0 in
#   spawn begin
#     ok() &
#     let a = try Some (input_line inchan) with End_of_file -> None in
#     close_in inchan ;
#     match a,snd (Unix.waitpid [] pid) with
#     | Some a,Unix.WEXITED 0 -> result(a)
#     | _,_ -> 0
#   end ;
#   wait, kill
# ;;
val create_arch : string -> (unit -> string) * unit Join.chan = <fun>
```

The function `create_arch` first performs the remote call by calling `rsh_arch`. The "run" function is the synchronous `wait` channel. As shown by its synchronization pattern, it transmits one result (one message on the internal channel `result`) to its caller. The message on `result` is sent by another concurrent agent, that reads one line from from the remote `arch` output, and then check the proper termination of the forked command (by `Unix.waitpid`). Additionally, a `kill` channel offers the ability to kill the forked Unix process. Finally, thanks to the internal channel `ok`, either `wait` will return a valid value, or the forked process will be destroyed as the result of a timeout.

1.10.3 Collecting results

We intend to execute `arch` on several remote machines, all those executions being performed concurrently. Hence, we need a way to collect n results produced concurrently. The task of collecting those results is performed by using the collectors of the standard library — see section 1.4.1.

```
# let collector n =
#   Col.create
#     (fun x xs -> match x with
#     | Some x -> x::xs
#     | None -> xs)
#     [] n
# ;;
```

```
val collector : int -> ('a option, 'a list) Col.t = <fun>
```

A collector is setup for collecting n values sent on channel `add`. Values may be worth collecting or not (`Some x` or `None`), but n messages must be sent on `add`, before the list of collected values can be returned by `wait`.

We now spawn n concurrent tasks, one per host in the list `hosts`.

```
# let archs hosts =
#   let col = collector (List.length hosts) in
#   List.iter
#     (fun host ->
#       let arch, kill = create_arch host in
#       spawn begin match timeout 1.0 arch () kill with
#       | Some a -> col.Col.collect(Some (host, a))
#       | None ->
#    prerr_endline ("Timeout for: "^host) ;
#    col.Col.collect(None)
#       end)
#   hosts ;
#  col.Col.wait()
# ;;
val archs : string list -> (string * string) list = <fun>
```

And here we go:

```
# archs ["saumur"; "beaune" ; "yquem" ; "macao"]
# ;;
Timeout for: macao
- : (string * string) list =
[("beaune", "alpha"); ("yquem", "i686"); ("saumur", "x86_64")]
```

Chapter 2

Distributed programming

This chapter presents the distributed and mobile features of JoCaml. JoCaml is specifically 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 sections), or they can be executed in a distributed manner on several machines.

In this section, we describe support for execution on several machines. To this end, we interleave a description of the model with a series of examples that illustrate the use of these primitives.

2.1 The Distributed Model

The execution of JoCaml programs can be distributed among numerous 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. 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 contents) without knowing whether this port name is locally- or remotely-defined. So far, 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 terminal 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.

An important issue when passing messages in a distributed system is whether the message contents is replicated or passed by reference. This is the essential difference between functions and synchronous channels. When a function is sent to a remote machine, its code and the values for its local variables are also sent there, and 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 and invocations on this name will forward the invocation to the machine were the name is defined, much

as in a remote procedure call. In the current implementation of JoCaml, passing a function in a distributed system is not yet implemented.

2.1.1 The name-server

Since JoCaml has lexical scoping, programs being executed on different machines 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 names and to build more complex communication patterns. To export names, the JoCaml library provides a name server (`Join.Ns`).

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. For instance, the following program contains two processes running in parallel. 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 begin
#   def f (x) = reply x*x to f in
#   Join.Ns.register Join.Ns.here "square" (f: int -> int);
#   0
# end
# ;;
- : unit = ()


# spawn begin
#   let sqr = (Join.Ns.lookup Join.Ns.here "square" : int -> int) in
#   print_int (sqr 2);
#   0
# end
# ;;
- : unit = ()
4
```

`lookup` and `register` functions are parameterized by a name server. Here, both processes are executed in the same runtime such that they have a direct access to the local name server (`Join.Ns.here`).

Communications through the name server are untyped. This weakness involves a good programming discipline.

2.1.2 Running several programs in concert

The runtimes that participate to a distributed computation are launched as independent executables, e.g. bytecode executables generated by the compiler and linked to the distributed runtime. a site (`Join.Site`) is associated to each runtime.

The following example illustrates a distributed computation with two machines. Let us assume that we have a single machine "`here.inria.fr`" that is particularly good at computing squares of integers; on this machine, we define a square function that also prints something when it is called (so that we can keep track of what is happening), and we register this function with key "`square`":

```
# def f (x) =
#   print_string ("["^string_of_int(x)^"] "); flush stdout;
#   reply x*x to f
# in Join.Ns.register Join.Ns.here "square" f
# ;;
- : unit

# let wait =
#   def x () & y () = reply to x
#   in x
# ;;
val wait : unit -> unit

# let main =
#   Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
#   wait()
# ;;
val main : unit
```

The function `Join.Site.listen` creates a socket waiting connections to the local site. Here, the server waits connections on the default Internet addresses of the host (`Join.Join.get_local_addr()` is the first returned by `Unix.gethostbyname`) on port `12345`. The call to the synchronous channel "`wait`" primitive tells the program to keep running after the completion of all local statements, so that it can serve remote calls.

On machine `here.inria.fr`, we compile the previous program (`p.ml`) and we execute it:

```
here> jocamlc p.ml -o p.out
here> ./p.out
```

We also write a program that relies on the previous machine to compute squares; this program first looks up for the name registered by `here.inria.fr`, then performs some computations and reports their results.

```
# let server =
#   let server_addr = Unix.gethostbyname "here.inria.fr" in
#   Join.Site.there (Unix.ADDR_INET(server_addr.Unix.h_addr_list.(0),12345))
# ;;
val server : Join.Site.t

# let ns = Join.Ns.of_site server
# ;;
val ns : Join.Ns.t

# let sqr = (Join.Ns.lookup ns "square": int -> int)
```

```
# ;;
val sqr : int -> int

# let log s x =
#   print_string ("q: "^s^" = "^string_of_int(x)^"\n"); flush stdout
# ;;
val log : string -> int -> unit

# let rec sum s n = if n = 0 then s else sum (s+sqr(n)) (n-1)
# ;;
val sum : int -> int -> int

# log "sqr 3" (sqr 3);
# log "sum 5" (sum 0 5)
# ;;
- : unit
```

This program first connects to `here.inria.fr:12345` with the function `Join.Site.there` to get the abstract value `server` which represents the JoCaml runtime on `here.inria.fr` and it gets the name server with the function `Join.Ns.of_site` (notice that there is the function `Join.Ns.of_sockaddr` to get directly the name server from the socket address). Then it defines `sqr` as the `square` channel of `here.inria.fr`. The `sum` function computes the sum of squares using the `sqr` function.

On another machine `there.inria.fr`, we compile and run our second program (`q.ml`):

```
there> jocamlc q.ml -o q.out
there> ./q.out
```

What is the outcome of this computation? Whenever a process defines new port names, this is done locally, that is, their guarded processes will be executed at the same place as the defining process. Here, every call to square in `sqr 3` and within `sum 5` will be evaluated as a remote function call to `here.inria.fr`. The actual localization of processes is revealed by the `print_int` statements: `f` (aliased to `sqr` on `there.inria.fr`) always prints on machine `here`, and `log` always prints on machine `there`, no matter where the messages are posted.

The result on machine `here.inria.fr` is:

```
[3] [5] [4] [3] [2] [1]
```

while the result on machine `there.inria.fr` is:

```
sqr 3= 9
sum 5= 55
```

2.1.3 Manage site termination

When a distributed application runs, some sites can terminate, fail or be unreachable. Hence, sending a message to a site which is not available on a synchronous channel raises the exception `Join.Exit`.

Let us define a function that test if a site is available or not. On the machine `here.inria.fr`, we define two synchronous channels: `living` that always returns `true` and `kill` that kills the JoCaml runtime.

```
# def living () = reply true to living
# ;;
val living : unit -> bool
```

```
# def kill () & wait () = reply to kill & reply to wait
# ;;
val kill : unit -> unit
val wait : unit -> unit
```

```
# let main =
#   Join.Ns.register Join.Ns.here "living" (living: unit -> bool);
#   Join.Ns.register Join.Ns.here "kill" (kill: unit -> unit);
#   Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
#   wait()
# ;;
val main : unit
```

On the machine `there.inria.fr`, we get the two synchronous channels defined on `here.inria.fr` and define a function `is_available` that calls `living`. If `living` returns a value then the JoCaml runtime on `here.inria.fr` is available. Otherwise, `living` raises the exception `Join.Exit`.

```
# let ns =
#   let server_addr = Unix.gethostbyname "here.inria.fr" in
#   Join.Ns.of_sockaddr (Unix.ADDR_INET(server_addr.Unix.h_addr_list.(0),12345))
# ;;
val ns : Join.Ns.t
```

```
# let living = (Join.Ns.lookup ns "living" : unit -> bool)
# and kill =  (Join.Ns.lookup ns "kill" : unit -> unit)
# ;;
val living : unit -> bool
val kill : unit -> unit
```

```
# let is_available () =
#   try living () with Join.Exit -> false
# ;;
val is_available : unit -> bool
```

```
# let main =
#   if is_available () then print_string "OK ! " else print_string "KO ! ";
#   kill ();
#   if is_available () then print_string "OK ! " else print_string "KO ! "
# ;;
val main : unit
```

The output of this program on `there.inria.fr` is:

```
OK ! KO !
```

The first time `is_available` is called, the JoCaml runtime on `here.inria.fr` is running. Then the call to `kill` stops the execution of this runtime such that the second call to `is_available` returns the value `false`.

Another way to deal with site termination is to use the function `Join.Site.at_fail`. This function records an asynchronous channel to call when a site fails. Let us define a simple example where two runtimes run on the same computer. The first runtime waits for some connections:

```
# let wait =
#   def x () & y () = reply to x
#   in x
# ;;
val wait : unit -> unit

# let main =
#   Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
#   wait()
# ;;
val main : unit
```

The second runtime gets the site listening on port `12345` and registers the channel `echo_failure` on the failure of this site.

```
# let server =
#   Join.Site.there (Unix.ADDR_INET(Unix.inet_addr_loopback, 12345))
# ;;
val server : Join.Site.t = <abstr>

# def echo_failure () =
#   print_string "FAILURE!";
#   print_newline();
#   0
# in
# Join.Site.at_fail server echo_failure
# ;;
- : unit = ()

# let wait =
#   def x () & y () = reply to x
#   in x
# ;;
val wait : unit -> unit = <fun>

# let main =
#   wait()
# ;;
val main : unit
```

If we execute these two runtimes and kill the first one (with `Ctrl-c` for example), then the second runtime prints `FAILURE!`.

2.2 A complete example

We want to build a very simple chat system where each user can broadcast a message to all the others.

2.2.1 Behavior of a client

The client is the part of the system executed by each user. The behavior of the client is to read messages from the user and broadcast them to the other clients.

```
# let read broadcast =
#   try
#     while true do
#       let msg = input_line stdin in
#       broadcast msg
#     done
#   with End_of_file -> ()
# ;;
val read : (string -> 'a) -> unit = <fun>
```

This function terminates when there are no more characters to read.

To implement the broadcast, we first have to define a type that represents a user and a data structure that allows to store the other users.

```
# type t_user =
#     { name : string;
#       site : Join.Site.t;
#       write : string -> unit; }
# ;;
type t_user = { name : string; site : Join.Site.t; write : string -> unit; }
# let get, add, remove =
#   def state(x) & get() = state(x) & reply x to get
#    or state(x) & add(user) = state(user::x) & reply to add
#    or state(x) & remove(s) =
#      state(List.filter (fun user -> not (Join.Site.equal user.site s)) x)
#   in
#   spawn state([]);
#   get, add, remove
# ;;
val get : unit -> t_user list = <fun>
val add : t_user -> unit = <fun>
val remove : Join.Site.t Join.chan = <abstr>
```

Each user is defined by his name, the site where he executes its client and the channel that allows to write on console of his client. To manipulate the set of users, there are the channels `get`, `add` and `remove`. `get` returns the list of the other users, `add` puts a new user into the set of users and `remove` remove the user executed on a given site. Notice that to test the equality of two sites, we have to use the function `Join.Site.equal` (we cannot use `=` and `==` may return a wrong value).

A simple way to broadcast a message is to send the message to each user in a sequential way.

```
# let seq_broadcast msg =
#   List.iter (fun user -> try user.write msg with _ -> ()) (get())
# ;;
val seq_broadcast : string -> unit = <fun>
```

We can also implement a parallel broadcast using `countdown` introduced section 1.4.1.

```
# let create_countdown n =
#   def count(n) & tick() = count(n-1)
#    or  count(0) & wait() = reply to wait in
#   spawn count(n) ;
#   tick,wait
# ;;
val create_countdown : int -> unit Join.chan * (unit -> unit) = <fun>
# let par_broadcast msg =
#   let others = get() in
#   let tick, wait = create_countdown (List.length others) in
#   List.iter
#     (fun user ->
#       spawn begin
#  begin try user.write msg with _ -> () end;
#  tick()
#       end)
#     others
# ;;
val par_broadcast : string -> unit = <fun>
```

Here, the broadcast terminates when all the messages are sent. An other solution that does not wait that all the messages are sent and that preserves the order of the messages is to use the buffer introduced section in 1.5.3. So, we define a function that replace the write channel of a user by a write channel with a buffer.

```
# type 'a buffer = { put : 'a -> unit ; get : unit -> 'a ; }
# ;;
type 'a buffer = { put : 'a -> unit; get : unit -> 'a; }
# let create_buffer () =
#   def alive (xs,y::ys) & get() = alive(xs,ys) & reply y to get
#   or alive(_::_ as xs,[]) & get() = alive([], List.rev xs) & reply get() to get
#   or alive(xs,ys) & put(x) = alive(x::xs,ys) & reply to put in
#   spawn alive([],[]) ;
#   {put=put; get=get;}
# ;;
val create_buffer : unit -> 'a buffer = <fun>
# let bufferize user =
#   let buff = create_buffer () in
#   def transmit() = user.write(buff.get()) ; transmit() in
#   spawn transmit ();
```

```
#   { user with write = buff.put }
# ;;
val bufferize : t_user -> t_user = <fun>
```

Bufferized users can be used with both broadcast definitions.

2.2.2 Add and remove users

We want to remove automatically a user when it is unreachable. So, we use the `Join.Site.at_fail` function introduced in section 2.1.3 to fired a channel that removes the unreachable user from the set of users.

```
# let at_fail user =
#   def remove_user() =
#     print_endline (user.name ^ " leaves the room ...");
#     remove user.site
#   in
#   Join.Site.at_fail user.site remove_user
# ;;
val at_fail : t_user -> unit = <fun>
```

Now, we define how to add a user to the set of users known by the client. This channel add the user to the set of clients and record the behavior to executes when the user is unreachable.

```
# def new_user(user_there) =
#   let user_there = bufferize(user_there) in
#   print_endline (user_there.name ^ " is in the room ...");
#   add user_there;
#   at_fail user_there;
#   0
# ;;
val new_user : t_user Join.chan = <abstr>
```

2.2.3 Connect two clients together

To create a connection, the clients need to exchange their user information (a value of type `t_user`). This value can be created as follows:

```
# let name_here = Unix.gethostname()
# ;;
val name_here : string = "saumur"
# def write_here(msg) = reply print_endline msg to write_here
# ;;
val write_here : string -> unit = <fun>
# let here =
#   { name = name_here;
#     site = Join.Site.here;
#     write = write_here; }
```

```
# ;;
val here : t_user = {name = "saumur"; site = <abstr>; write = <fun>}
```

Here, we define only one write channel on the client. This channel will be shared by all the other users. In this case, we cannot distinguish the user that sends the message on this channel.

An other solution is to create one channel by user and prefix each message by the name of the user:

```
# let write_from name =
#   def write_here(msg) =
#     reply print_endline (name^"> "^msg) to write_here
#   in
#   write_here
# ;;
val write_from : string -> string -> unit = <fun>
# let make_here name_there =
#     { name = name_here;
#       site = Join.Site.here;
#       write = write_from name_there; }
# ;;
val make_here : string -> t_user = <fun>
```

Let's see now, how two clients can exchange their user information. Suppose that a client `B` wants to build a symmetric connection with a client `A`: `A` must be added to the set of user of `B` and `B` must be added to the set of user of `A`. The protocol is the following. `B` sends its name (`name_there`) and a channel (`connect_there`) to `A` through a `listen` channel on `A`. Then `A` uses the `connect_there` channel to send back to `B` its user information and `A` obtains in return the `B` user information.

So the `listen` channel (on `A`) and the `connect` channel (on `B`) are defined as follows:

```
# def listen(name_there, connect_there) =
#   let here =
#     { name = name_here;
#       site = Join.Site.here;
#       write = write_from name_there; }
#   in
#   let user_there = connect_there(here) in
#   new_user(user_there)
# ;;
val listen : (string * (t_user -> t_user)) Join.chan = <abstr>
# def connect_here(user_there) =
#   let here =
#     { name = name_here;
#       site = Join.Site.here;
#       write = write_from user_there.name; }
#   in
#   reply here to connect_here
```

```
#   & new_user(user_there)
# ;;
val connect_here : t_user -> t_user = <fun>
```

The `listen` channel is exported through the name server to allow to create some connections.

```
# type t_connect = t_user -> t_user
# ;;
type t_connect = t_user -> t_user
# type t_listen = (string * t_connect) Join.chan
# ;;
type t_listen = (string * t_connect) Join.chan
# let () = Join.Ns.register Join.Ns.here "listen" (listen: t_listen)
# ;;
```

So, to build a connection from a site identity we define the channel `connect_to` as follows.

```
# def connect_to(site) =
#   let ns = Join.Ns.of_site site in
#   let listen_there = (Join.Ns.lookup ns "listen" : t_listen) in
#   listen_there(name_here, connect_here)
# ;;
val connect_to : Join.Site.t Join.chan = <abstr>
```

2.2.4 Build a chat room

The last point to build a chat room is to know the sites that participate to the chat. We use a centralize approach where each site register to a server and get in return the list of sites already registered.

Let's define the server part:

```
# let register =
#   def state(xs) & get_and_add(x) =
#       state(x::xs)
#     & reply xs to get_and_add
#     & begin Join.Site.at_fail x (def rm() = remove(x) in rm); 0 end
#    or state(xs) & remove(x) =
#       state(List.filter (fun x' -> not (Join.Site.equal x x')) xs)
#   in
#   spawn state [];
#   get_and_add
# ;;
val register : Join.Site.t -> Join.Site.t list = <fun>
# let server () =
#
Join.Ns.register Join.Ns.here "register" (register: Join.Site.t -> Join.Site.t list);
#   Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
```

```
#   def wait() & abs() = reply to wait in wait()
# ;;
val server : unit -> unit = <fun>
```

The `register` channel defines a shared data structure that store the sites registered. A call to `register` adds a new site to the data structure and returns the previous state. Sites unreachable are automatically remove from the set of sites thanks to the behavior recorded by `Join.Site.at_fail`.

The `server` function exports the `register` channel, does the call to `Join.Site.listen` and wait forever.

We define now the client part:

```
# let client server_site =
#   let ns = Join.Ns.of_site server_site in
#
let register = (Join.Ns.lookup ns "register" : Join.Site.t -> Join.Site.t list) in
#   let others = register Join.Site.here in
#   List.iter (fun site -> spawn connect_to site) others;
#   read par_broadcast
# ;;
val client : Join.Site.t -> unit = <fun>
```

The client is parameterized by the site of the server. Its behavior is to get the `register` channel of the server. Then to register and get the list of the other sites. Try to connect to these sites and to executes the `read` function.

At last, the main function choose the behavior to execute depending on the command line options.

2.3 Limitations

The actual implementation of JoCaml has some limitations. Most of them comes from the messages marshaling.

2.3.1 Code mobility

Code mobility is not yet implemented. It means that a function can not be sent on a distributed channel. We illustrate this point with a JoCaml program `server.ml` which awaits a function and a value and then does the application.

```
# def apply (f, x) = reply f x to apply in
# Join.Ns.register Join.Ns.here "server" (apply: ((int -> int) * int -> int))
# ;;
- : unit = ()

# let wait =
#   def x () & y () = reply to x
```

```
#   in x
# ;;
val wait : unit -> unit = <fun>

# let main =
#   Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
#   wait()
# ;;
val main : unit = ()
```

Then we define the following `client` program which connects to the server and sends a function:

```
# let ns =
#   Join.Ns.of_sockaddr (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345))
# ;;
val ns : Join.Ns.t = <abstr>

# let main =
#   let compute = (Join.Ns.lookup ns "server" : ((int -> int) * int -> int)) in
#   print_int (compute((fun x -> x + 1), 2))
# ;;
val main : unit = ()
```

If we try to run this example, we obtain a runtime error:

```
here> jocamlc server.ml -o s.out
here> ./s.out

there> jocamlc client.ml -o c.out
there> ./c.out
Fatal error: exception Invalid_argument("output_value: functional value")
```

2.3.2 References and exceptions

The behavior of a channel can change if it is called from the same JoCaml runtime or not. We first define a simple synchronous channel who returns its message.

```
# def id (x) = reply x to id
# ;;
val id : 'a -> 'a = <fun>
```

Then we send a mutable value on this channel from the same runtime:

```
# let r = ref 21
# ;;
val r : int ref = {contents = 21}
# let r' = id (r)
# ;;
val r' : int ref = {contents = 21}
# r := !r * 2;
```

```
# print_int !r'
# ;;
- : unit = ()
42
```

We can observe that `r` and `r'` are the same reference.

Now, we execute the same program except that the `id` function is computed in an other runtime:

```
# let ns =
#   Join.Ns.of_sockaddr (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345))
# ;;
val ns : Join.Ns.t = <abstr>

# let id = (Join.Ns.lookup ns "id" : (int ref -> int ref))
# ;;
val id : int ref -> int ref = <fun>

# let r = ref 21
# ;;
val r : int ref = {contents = 21}

# let r' = id (r)
# ;;
val r' : int ref = {contents = 21}

# r := !r * 2;
# print_int !r'
# ;;
- : unit = ()
21
```

Here, we can see that `r'` is not modified by `r := !r * 2` because `r` and `r'` are two different reference cells. When a mutable value go through the network, a new copy is created.

We can notice that we have the same behavior for exceptions.

```
# let ns =
#   Join.Ns.of_sockaddr (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345))
# ;;
val ns : Join.Ns.t = <abstr>

# let id = (Join.Ns.lookup ns "id" : (exn -> exn))
# ;;
val id : exn -> exn = <fun>

# exception E
# ;;
exception E

# try raise (id (E)) with
# | E -> print_string "E is raised"
# | _ -> print_string "An exception is raised"
# ;;
```

```
- : unit = ()
An exception is raised
```

Another, similar, more frequent, situation occurs when the remote site raises an exception as an answer to a synchronous call. When all executions take place in the same runtime, everything is fine.

```
# def f () = let _x = raise E in reply to f
# ;;
val f : unit -> unit = <fun>
# try f () with
#   | E -> print_string "E is raised"
#   | _ -> print_string "An exception is raised"
#   ;;
- : unit = ()
E is raised
```

But now assume the following code on one runtime `A`, whose name service is used.

```
# (* Runtime A *) exception E
# ;;
exception E
# def raise_E () = let _x = raise E in reply to raise_E
# ;;
val raise_E : unit -> unit = <fun>
# Join.Ns.register Join.Ns.here "raise_E" (raise_E : unit -> unit)
# ;;
- : unit = ()
```

On runtime `B`, we also define exception `E`, retrieve channel `raise_E` from `A`, and then send a synchronous message on it.

```
# (* Runtime B *) exception E
# ;;
exception E
# let raise_E = (Join.Ns.lookup ns "raise_E" : unit -> unit)
# ;;
val raise_E : unit -> unit = <fun>
# try raise_E() with
# | E -> print_string "E is raised"
# | _ -> print_string "An exception is raised"
# ;;
- : unit = ()
An exception is raised
```

This behavior can be sumarized by saying that exceptions are *generative*: the execution of `exception E` yields a unique exception and executing `exception E` twice (moreover on two different runtimes) yields two different exceptions, which happen to have the same name `E`. Maybe we can live with such a notion. Unfortunately, this is not the whole story, when they travel from one runtime to another exeception are copied, which contradicts generativity. Consider a similar example, where another synchronous channel is defined on `A` as follows:

```
# (* Runtime A *)
# def raise_e(e) = let _x = raise e in reply to raise_e
# ;;
val raise_e : exn -> unit = <fun>
# Join.Ns.register Join.Ns.here "raise_e" (raise_e : exn -> unit)
# ;;
- : unit = ()
```

While, on `B` we define the exception `E`, retrieve `raise_e` from the name service of `A` and send `E` on channel `raise_e`.

```
# (* On runtime B *)
# exception E
# ;;
exception E
# let raise_e = (Join.Ns.lookup ns "raise_e" : exn -> unit)
# ;;
val raise_e : exn -> unit = <fun>


# try raise_e(E) with
#  | E -> print_string "E is raised"
#  | _ -> print_string "An exception is raised"
# ;;
- : unit = ()
An exception is raised
```

And here, although the remote site `A` apparently raises an exception defined on site `B`, it in fact raises a copy of it. This copy is then copied once more while transmitted back to site `B`. Since exception matching is performed by using pointer equality, it makes the difference beetween those two copies.

A specific mecanism somehow solves the problem of exceptions raised by remote runtimes. One may apply the mecanism to any pre-existing exception by the declaration `def exception`, for instance on runtime `B`.

```
# (* Runtime B *)def exception E
# ;;
```

Then, we can try our two examples again, still on runtime `B`.

```
# try raise_E() with
#  | E -> print_string "E is raised"
#  | _ -> print_string "An exception is raised"
# ;;
- : unit = ()
E is raised
# try raise_e(E) with
#  | E -> print_string "E is raised"
#  | _ -> print_string "An exception is raised"
```

```
# ;;
- : unit = ()
E is raised
```

And now, both the copy of the other `E` defined on site `A` (`raise_E`) and the copy of copy of `E`, match `E` in the exception handlers.

There is no miracle here, the JoCaml runtime system of `B` intercepts exceptions *raised*[1] by `A` and makes all exceptions whose name are `E` become an unique exception. As a result there is only one exception `E` on runtime `B`. Observe that the `def exception` construct is a non-trivial semantical change over Objective Caml: it more or less introduces matching of exception by structure, but only for exceptions raised from one runtime to another. Be cautious. Notice that applying `def construct` twice to the same exception yields a fatal error.

By default, JoCaml runtimes share all built-in exception (such as `Not_found`, `Invalid_argument`, etc.) and the `Join.Exit` exception.

2.3.3 Typing

Communications through the name service are not typed. In the following example we define a synchronous channel of type `int -> int` and register it on the name service.

```
# def f (x) = reply x+1 to f
# ;;
val f : int -> int = <fun>
# Join.Ns.register Join.Ns.here "incr" (f: (int -> int))
# ;;
- : unit = ()
```

Then, we retrieve the channel and use it with type `float -> float`.

```
# let g = (Join.Ns.lookup Join.Ns.here "incr" : (float -> float))
# ;;
val g : float -> float = <fun>
# print_float (g 0.5);;
- : unit = ()
9.35959773756e-232
```

We obtain an indeterministic value and we do not have type error! Notice that in most situations JoCaml will crash. A good programming discipline is to define shared types in a separate file and to annotate the functions `Join.Ns.register` and `Join.Ns.lookup` with these types.

[1] Exceptions transmitted as ordinary messages are not intercepted

Part II

User Manual

Chapter 3

The JoCaml language

This document is intended as a reference manual for the JoCaml language. JoCaml is an extension of Objective Caml, and this manual addresses only the constructs which are new with respect to Objective Caml, introducing syntax and informal semantics for the new constructs. A good working knowledge of Objective Caml is assumed.

Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (`like this`). Non-terminal symbols are set in italic font (*like that*). Square brackets $[\ldots]$ denote optional components. Curly brackets $\{\ldots\}$ denotes zero, one or several repetitions of the enclosed components. Curly bracket with a trailing plus sign $\{\ldots\}^+$ denote one or several repetitions of the enclosed components. Parentheses $(\ldots)$ denote grouping.

3.1 Lexical issues

JoCaml has the same lexical conventions as Objective Caml, with additional keywords `spawn`, `def` and `reply`.

Keywords `or` and `&` exist in both languages, but with very different meanings. As a consequence, JoCaml users cannot use `or` and `&` for the boolean connectors "or" and "and". Notice that this practice is deprecated in Objective Caml: to express boolean connectors one should (in Objective Caml) and must (in JoCaml) use the operators `||` and `&&`.

3.2 Values

JoCaml has an additional kind of values : channels or port names. Channels have a sending side and a receiving side and can be seen as carrying messages from one side to the other. Channels come in two flavors: asynchronous and synchronous. Sending a message on an asynchronous channel always succeeds, but this tells little about whether the message is received or not at the other side of the channel. Synchronous channels behave like functions.

3.3 Types

JoCaml has an additional primitive type : the type *typexpr* `Join.chan` of asynchronous channels that carry values of type *typexpr*. The type is primitive in the sense that compiler knows about it. Synchronous channels are typed as functions.

3.4 Expressions

expr ::= *ocaml-expr*
| `spawn` *process*
| *join-definition* `in` *expr*

Objective Caml expressions are extended with the expression `spawn` *process* which evaluates to `()` and executes *process* asynchronously. Local definition of channels by a *join-definition* is possible inside expressions.

3.5 Processes

process ::= `0`
| *process* `&` *process*
| *expr expr*
| *expr* `;` *process*
| `reply` [*expr*] `to` *lowercase-ident*
| `if` *expr* `then` *process* [`else` *process*]
| `match` *expr* `with` {`|` *ocaml-pattern* [`when` *expr*] `->` *process*}$^+$
| `let` [`rec`] *ocaml-let-binding* {`and` *ocaml-let-binding*} `in` *process*
| `for` *lowercase-ident* `=` *expr* (`to` | `downto`) *expr* `do` *process* `done`
| `(` *process* `)`
| `begin` *process* `end`
| *join-definition* `in` *process*

The class of processes is the main syntactical extension of JoCaml with respect to Objective Caml. Processes have no result. By convention, we say that processes are *executed*, while expressions are evaluated to some result. Moreover, expressions may fail to produce a result and instead raise an exception. The matching concept for processes is *abnormal termination*. When a process terminates abnormally, the JoCaml runtime system issues a warning message.

3.5.1 Basic processes

Those are directly inspired from the join-calculus.

Inert process

The simplest process is `0` (concrete syntax is the digit `0`). This process does nothing.

Grouping

The processes `(` *process* `)` and `begin` *process* `end` execute as *process* does. Both constructs are semantically equivalent, using one construct or the other is a matter of style.

Asynchronous send

Asynchronous send $expr_1$ $expr_2$ evaluates $expr_1$ to an asynchronous channel and $expr_2$ to a value. Then, the value is sent on the channel. Notice that the syntax is the one of function application, but in process context. Although the syntax is quite general, common usage is simpler. Most often, $expr_1$ is a channel name, while $expr_2$ is a tuple of expressions : *lowercase-ident* `(` $expr_1$, $expr_2$, ..., $expr_k$ `)`.

Concurrent execution

The operator `&` is the parallel composition. Executing $process_1$ `&` $process_2$ amounts to executing $process_1$ and $process_2$ concurrently.

Reply to synchronous sends

The process `reply` *expr* `to` *lowercase-ident* sends the value of *expr* as a reply on the synchronous channel *lowercase-ident*, which must be the name of a synchronous channel. The `reply` construct is of course to be used while defining the receiving side of a synchronous channel. On the other side, sending a message on a synchronous channel is very similar to calling a function, and the replied *expr* is like the value returned by a function call.

There are severe linearity constraints over the usage of `reply`. Linearity constraints here mean that the compiler enforces that exactly one reply to a given channel is allowed. For instance, in $process_1$`&`$process_2$, the processes $process_1$ and $process_2$ must reply to disjoint sets of channels; while, in `if`*expr* `then`$process_1$ `else`$process_2$, the processes $process_1$ and $process_2$ must reply to the same channels.

3.5.2 Composed processes

Composed processes are the equivalent of some of the control structures of Objective Caml, but at the process level.

Sequence

The process *expr* `;` *process* evaluates *expr* first, then it executes *process*. The expression must be of unit type (by contrast with Objective Caml). Notice that the item before the semicolon `;` is an expression, not a process — a process there would be meaningless.

Operator `&` binds tighter than operator `;`. As a result, *expr*`;`$process_1$ `&`$process_2$ is legal syntax and *expr* evaluates before both $process_1$ and $process_2$ execute. Moreover, $process_1$`&`*expr* `;`$process_2$ is illegal syntax.

Conditional

Executing the process `if` *expr* `then` $process_1$ `else` $process_2$ executes $process_1$ if *expr* evaluates to the boolean `true`, and $process_2$ if $expr_1$ evaluates to the boolean `false`.

The `else` $process_2$ part can be omitted, in which case it defaults to `else 0`.

Pattern matching and `let` definitions in processes

Those two constructs are like their (Objective Caml) expression counterpart, except that they are processes and that they introduce bindings of names to values over processes in place of expressions.

Concurrent `for` loop

The process equivalent of the traditional `for` loop. Iterations are executed concurrently.

Local definition of channels in processes

Local definition of channels by a *join-definition* is possible inside processes.

3.6 Join definitions

Join definitions reflect the key idea behind the join-calculus. They define channels, bind them to names and define the receptors on channels.

join-definition	::=	`def` *reactions* {`and` *reactions*}
reactions	::=	*reaction* {`or` *reactions*}
reaction	::=	*join-pattern* `=` *process*
join-pattern	::=	*channel-decl* {`&` *channel-decl*}
channel-decl	::=	*lowercase-ident* `(` *ocaml-pattern* `)`

3.6.1 Informal semantics

The channels defined and bound are the set of *lowercase-ident* in *channel-decl* above. Channel names cannot be repeated inside the *join-pattern* of a given *reaction*, but can appear in several *reaction* in a *reactions*. A *reaction* define both a synchronization behavior as its *join-pattern* and a receptor as its *process*. More precisely, when messages are present on all the channels of a given *reaction* and that their values match the corresponding formal arguments *ocaml-pattern* above, then the reaction is *active* and the guarded *process may* be executed. A *reactions* is a list of competing *reaction*, as suggested by the keyword `or`. Implementation offers the limited guarantee that if exactly one reaction in a list of competing reactions is active, then the execution of its guarded process starts. If several reactions are active at the same time, then one of them is selected, which one being left unspecified purposely.

For example, the construct

```
def   ident1(patt1 )                      =   process1
 or   ident1(patt1 ) &ident2 (patt2 )     =   process2
 in   process
```

defines two channels ($ident_1$ and $ident_2$) locally to *process*. These channels are defined by two join patterns. The first join patter n matches when there is a message on the channel $ident_1$ and that the content of the message matches the pattern $patt_1$. The second join pattern matches when there is a message on both channels and the content of each message matches its pattern. When a message is sent to the channel $ident_1$, if one of the two join pattern matches, its associated process is executed. If the two join patterns match, one of the two process is executed.

3.6.2 Scopes

The scope of the channel names defined by *join-definition* extends to all receptors (*process* in *reaction*). In other words, channel definitions are recursive by default. There may be several sets of competing behaviors (several *reactions*) in a *join-definition*, separated by the keyword `and`. Then, the set of channel names defined by the various *reactions* must be pairwise disjoint.

Moreover, in a reaction *join-pattern* `=`*process*, the scope of the variables inside the formal arguments (*ocaml-pattern* in *channel-decl*) extends to *process*. Those are bound by following the usual rules of the pattern matching of Objective Caml. All such variables must be pairwise distinct in a given *join-pattern*.

3.6.3 Asynchronous and synchronous channels

A reaction *join-pattern* `=`*process* defines a channel *ident* to be synchronous when `reply...to`*ident* occurs inside *process*. The `reply` construct can be seen as an asynchronous message sending on some implicit "continuation" channel. The scope of this continuation channel does not extend over `def` join definition that occur inside *process* above. If another *reaction* defines the channel *ident* in the set of competing behaviors *reactions*, then *ident* must also be defined as synchronous there.

3.7 Module expressions

Definitions in structures (modules) are extended as follows:

definition ::= *ocaml-definition*
| *join-definition*
| `def exception` *constr-name*

Toplevel definition in structure can of course be *join-definition*. The additional `def exception`*exception* constructs enables matching by structure in `try ... with ...` handlers. The construct, somehow aleviates the burden of having generative exceptions in a distributed context. The name *exception* must be the name of a pre-existing exception, and `def exception`*exception* must be executed at most once. See Section 2.3.2 for an example.

Chapter 4

JoCaml Tools

4.1 A few words on implementation(s)

The JoCaml system is structured as the Objective Caml system, with a bytecode compiler `jocamlc` and a native-code compiler `jocamlopt`. In fact, JoCaml is a modified Objective Caml. The JoCaml compilers translate Join constructs into function calls to an ad-hoc library. The ad-hoc library is an extension of the thread library of Objective Caml, which we simply call the `threads` library. The `threads` library comes in two flavors: virtual-machine level threads and system-level threads. JoCaml uses system-level threads by default. As a result of its design, JoCaml inherits the limitations of Objective Caml thread implementations. In particular, there can be at most one thread actually executing at a time, even on multi-core machines.

The compilers `jocamlc` and `jocamlopt` should be binary compatible with Objective Caml. More precisely, while installing JoCaml, a "companion Objective Caml" is defined. The companion system must have the same version number as the JoCaml system. Then, the search paths for the compiler are completed by the ones of the companion system. For that reason JoCaml distribution includes a limited subset of Objective Caml libraries.

4.2 Summary of tool modifications

All JoCaml tools are Objective Caml tools with an initial "j": `jocamlc` `jocaml` etc.

`jocamlc` The bytecode compiler. By default, `jocamlc` assumes compilation with respect to the system threads library. In that aspect, it behaves like `ocamlc` with command line options `-thread`, with libraries `unix.cma` and `threads.cma` added at link time.

New or modified options are:

`-v` Print the version number of the compiler, the location of the standard library directory and the location of the companion Objective Caml, if any; then exit.

`-nojoin`
Act as an Objective Caml compiler as much as possible:

- Do not recognize the keywords that are specific to JoCaml (`def`, `spawn` and `reply`).
- Cancel the effect of `-thread` and `-vmthread`.

- Suppress implicit link arguments `unix.cma` and `thread.cma`.

This option is useful mostly for system bootstrap.

`-thread` **(resp.** `-vmthread`**)**
Compile or link multithreaded programs, in combination with the system (resp. virtual-machine level) threads library. By contrast with `ocamlc`, `-thread` is the default.

`jocaml` The interactive toplevel. By defaults this toplevel includes the system-level threads library and the unix library.

`jocamlrun` The JoCaml virtual machine that executes bytecode files produced by `jocamlc`. C shared library are searched as `ocamlrun` does with an additional step beetween step 4. and 5.

4-bis. Directories specified in the file `ld.conf` of companion Objective Caml standard library directory, if any.

`jocamlopt` The native-code compiler. Added or modified options are the same as for `jocamlc`, except for the `-vmthread` option which does not exist.

`jocamllex` **and** `jocamlyacc` Lexer and parser generators. Those two tools are unchanged, except for their names.

`jocamldep` The dependency generator. The `jocamldep` progam accepts JoCaml sources as input. This default behavior can be changed by the following option:

`-nojoin`
Act over pure Objective Caml files.

`jocamlmklib` The tool to build mixed C/JoCaml libraries. There are two additional (cosmetic) options:

`-jocamlc` *cmd*
Alias for `-ocamlc` *cmd*.

`-jocamlopt` *cmd*
Alias for `-oocamlopt` *cmd*.

`jocamlmktop` The tool for building custom toplevels. The produced toplevels accept and execute JoCaml source code.

Chapter 5

The JoCaml library

JoCaml core library `Join` provides the built-in types for channels and sites, and basic operations on those. Some additional modules are provided that capture common JoCaml programming patterns. All the following modules are automatically linked with the user's object code files by the JoCaml compilers (Chapter 4).

5.1 Module `Join` : The JoCaml core library.

This module offers basic functionalities for JoCaml.

`type -'a chan`

The type of asynchronous channels carrying values of type `'a`.

`exception Exit`

Raised by the JoCaml runtime system, when some remote synchronous call cannot be completed because of the failure of the remote site

`val exit_hook : unit -> unit`

Hook to be given as argument to `Pervasives.at_exit`. This will somehow control termination of program. More precisely, program terminates when they is no more work to achieve. This does not apply to program engaged in distribution.

See also `Pervasives.at_exit`.[`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#VALat_exit`]

`type 'a debug = string -> ('a, unit, string, unit) Pervasives.format4 -> 'a`

The type of the argument of `Join.debug`[5.1]

`val debug : 'a debug`

Print a message on standard error. Usage: debug tag fmt ...

- fmt ... is in printf style.
- tag is a string. A lock is taken so that messages do not interleave.

See also The `Printf` module.[`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Printf.html`]

Site definition

```
module Site :
  sig
```

Sites are abstractions of JoCaml runtimes. Sites have unique identities, which can be passed on channels, computed from Internet addresses or extracted from asynchronous channels.

Sites must be compared with the `equal` and `compare` functions of this module. Sites are useful for performing crude failure detection (See `Join.Site.at_fail`[5.1] below).

```
type t
```

The type of site identities.

```
val here : t
```

Local site identity.

```
val there : Unix.sockaddr -> t
```

Get identity of the remote site listening on sockaddr. Raise `Failure` if the connection cannot be established.

```
val listen : Unix.sockaddr -> unit
```

Start to listen for connections on the socket address given as argument. Raises `Failure` in case of failure.

```
val where_from : 'a Join.chan -> t
```

`where_from c` returns the identity of the remote site where reception on channel `c` takes place.

```
val equal : t -> t -> bool
```

Test the equality of two sites.

```
val compare : t -> t -> int
```

Compare two sites, order is arbitrary.

```
val at_fail : t -> unit Join.chan -> unit
```

`at_fail s c` registers channel `c` as a guard on failure of site `s`. If `s` failure is detected, a message () is sent on channel `c`.

At the moment, site failure detection is a bit unsafe, due to naive routing. A failure may express the impossibility to contact a remote site for the first time.

```
val get_local_addr : unit -> Unix.inet_addr
```

Returns the default Internet address of the local site, never fails. At worst, `get_local_addr ()` returns the loopback address `Unix.inet_addr_loopback`

`end`

Dynamic, unsafe, value repository.

```
module Ns :
  sig
```

Dynamic, unsafe, value repository.

Every site offers a name service. The name service provides a mapping from strings to values.

`type t`

Abstract type for the name service.

`val here : t`

The local name service.

`val of_site : Join.Site.t -> t`

Get remote name service by site identity.

`val there : Unix.sockaddr -> t`

Get remote name service by socket address. Basically, `there addr` is `of_site (Site.there addr)`.

`val of_sockaddr : Unix.sockaddr -> t`

Synonym for `there`

`val lookup : t -> string -> 'a`

Find value, raise `Not_found` when not present, or `Join.Exit`[5.1] if attemping to lookup on a failed remote name service.

`val register : t -> string -> 'a -> unit`

Register binding, returns when done. Raise `Join.Exit`[5.1] if attempting to register on a failed remote name service.

`end`

5.2 Module `JoinCount` : Counting n asynchronous events

The following submodules are succesive refinements of the *count n events* programming idiom. Here an event is a message sent on an asynchronous channel.

- `JoinCount.Down`[5.2] just counts n messages sent on a `tick` channel.
- `JoinCount.Collector`[5.2] additionnaly computes a resut from the nessages sent on a `collect` channel.
- `JoinCount.Dynamic`[5.2] is a refinment of `Collector` where n is not known in advance.

```
module Down :
  sig
```

```
    type ('a, 'b) t = {
      tick : unit Join.chan ;
      wait : unit -> unit ;
    }
    val create : int -> ('a, 'b) t
```

`create n` returns a countdown `c` for `n` events. That is, after `n` messages on `c.tick`, the call `c.wait()` will return. Observe that at most one call `c.wait()` returns.

```
  end
```

Simple countdowns.

```
module Collector :
  sig
```

```
    type ('a, 'b) t = {
      collect : 'a Join.chan ;
      wait : unit -> 'b ;
    }
```

Type of collectors.

Collectors are refinements of countdowns, which collect and combine `n` partial results (type `'a`) into a final result (type `'b`). Given a collector `c` for `n` events, with combining function `comb` and initial result `y0`:

- The `n` events, `x1`,...,`xn`, are sent as `n` messages on `c.collect`. Notice that the notation `xi` does not imply any kind of ordering.
- Then `c.wait()` returns the value `comb x1 (comb x2 (... (comb xn y0)))`. Again, at most one call `c.wait()` returns.

```
    val create : ('a -> 'b -> 'b) -> 'b -> int -> ('a, 'b) t
```

`create comb y0 n` returns a collector of `n` events of type `'a`, with combining function `comb` and initial result `y0`.

```
  end
```

Collecting countdowns, or *collectors*.

```
module Dynamic :
  sig
```

```
    type ('a, 'b) t = {
      enter : unit -> unit ;
      leave : 'a Join.chan ;
      wait : unit -> 'b ;
      finished : unit Join.chan ;
    }
```

Dynamic collectors are refinement of simple collectors, for which the number of events to collect need not be given in advance.

Given a dynamic collector `c`, defined as `create comb y0`:

- `c` is informed of the future occurence of an event `xi`, by sending an unit message on `c.enter()`.
- `c` is informed of the occurence of event `xi` by sending a message `c.leave(xi)`.
- `c` is informed that no more event will occur by sending a message `c.finished()`.

Then the call `c.wait()` will return the combined result `comb x1 (comb x2 (... (comb xn y0)))`, once all the announced events have occured. Observe that at most one such call is allowed.

```
    val create : ('a -> 'b -> 'b) -> 'b -> ('a, 'b) t
```

`create comb y0` returns a dynamic collector of events of type `'a`, with combining function `comb` and initial result `y0`.

```
  end
```

Dynamic collectors

5.3 Module `JoinFifo` : Concurrent fifo buffers.

Concurrent fifo's offer blocking `get` operations. More precisely, `get` operations on, active, empty fifo's are blocking.

Fifo behavior can be observed in the simple situation where one agent (*producer*) is putting elements, while another agent (*consumer*) is getting them. Then the following guarantees hold:

- The consumer see the elements in producing order.

- If the producer closes the fifo, and does not put any additional elements, then the consumer will have retrieved all elements when the call to `close` returns.

```
type 'a t = {
  put : ('a * bool Join.chan) Join.chan ;
```
Put element into fifo.
```
  get : 'a option Join.chan Join.chan ;
```
Get element from fifo.
```
  close : unit -> unit ;
```
Close fifo, returns only when the fifo is empty
```
  kill : unit Join.chan ;
```
Close fifo, returns immediately, discarding any pending element.

}

The type of fifo buffer.

```
val create : unit -> 'a t
```
Create a new fifo buffer.

Interface to concurrent fifo's is mostly asynchronous. Let `f` be a fifo.

- `f.put(x,k)` put `v` into the fifo `f`. The channel `k` receives a (boolean) message `b`, where:
  - If `b` is `true`, then `v` was succesfully entered into `f`.
  - If `b` is `false`, then `v` could not be added to `f`. That is, `f` have been closed or killed.
- `f.get(k)` retrieve one element from the fifo, The channel `k` receives a (`'a option`) message, where:
  - `None` expresses that the fifo is closed.
  - `Some x` expresses that element `x` is retrieved from the fifo.

Operations `close` and `kill` both close the fifo, but with different behaviors as regards non-empty fifos.

- `f.close()` waits for the fifo to be empty before closing it and returning.
- `f.kill()` is an asynchronous channel, sending a message on `f.kill` closes the fifo immediately.

In the producer/consumer scheme, `f.close` is for the producer to signal the end of produced elements; while `f.kill` is for the consummer to signal that it will not accept more elements.

Producer/consumer interface

```
module P :
  sig
    type 'a t = {
      put : ('a * bool Join.chan) Join.chan ;
      close : unit -> unit ;
    }
  end
```

Type of producers

```
module C :
  sig
    type 'a t = {
      get : 'a option Join.chan Join.chan ;
      kill : unit Join.chan ;
    }
  end
```

Type of consumers

```
val create_prod_cons : unit -> 'a P.t * 'a C.t
```

Create a pair of producer/consumer connected by a fifo. Producer have the put and close operations, while consumer have the get and kill operations.

Synchronous interface

Type of fifo with synchronous operations

```
module S :
  sig
    exception Closed
    type 'a t = {
      put : 'a -> unit ;
      get : unit -> 'a ;
      close : unit -> unit ;
      kill : unit -> unit ;
    }
  end
```

```
val create_sync : unit -> 'a S.t
```

Records of type 'a S.t offer the same fields as the ones of type `'a t`, but they hold synchronous channels (of functional type)

- `put` of type `'a -> unit` either returns (when successful) or raise the exception `S.Closed`.
- `get` follows the same behavior.
- `close` of type `unit -> unit` closes the fifo, returning when the fifo is empty.

5.4 Module `JoinProc` : Convenience functions for forking Unix commands.

All functions fork commands given in the style of the `Unix.execvp` function. That is, a command is a program name plus an array of command line arguments and the program name is searched in path.

Functions return the pid of the child process that executes the program and some I/O channels, exactly which ones depends on the function.

```
val command : string -> string array -> int
```

`command prog args` executes program `prog` with arguments `args` in a child process. Standard channels stdin, stdout, and stderr are the ones of the parent process

```
val open_in : string -> string array -> int * Pervasives.in_channel
```

Same as `JoinProc.command`[5.4] above, except that the forked process standard output is redirected to a pipe, which can be read via the returned input channel.

```
val open_out : string -> string array -> int * Pervasives.out_channel
```

Same as `JoinProc.command`[5.4] above, except that the forked process standard input is redirected to a pipe, which can be written to via the returned output channel.

```
val open_in_out :
  string ->
  string array -> int * (Pervasives.in_channel * Pervasives.out_channel)
```

Redirects both standard output and input of the forked command to pipes. Returns `pid,(outch,inch)`, where `outch` is for reading the forked command standard output, and `inch` is for writing the forked command standard input

```
val open_full :
  string ->
  string array ->
  int *
  (Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel)
```

Redirects all three standard channels of the forked command to pipes. Returns `pid,(outch,inch,errch)`, where `outch` and `errch` permit reading the forked command standard output and standard error respectively, while `inch` permits writing on the forked command standard input.

Some of Objective Caml modules are included in the JoCaml distribution.

- The core library `Pervasives`.
- The whole standard library.
- The `Unix` library. By contrast with Objective Caml, users programs are linked with the `Unix` library by default.
- The threads libraries are present in a different form. By contrast with Objective Caml, users programs are linked with one of threads library by default. All the functionalities of Objective Caml threads are present. However, users should probably refrain from creating threads explicitely.
- The `Graphics` library.
- The `Dynlink` library.

Other Objective Caml libraries are available through a companion Objective Caml system (see Section 4.1).