The Join-Calculus language release 1.01

Documentation and user's manual

Cédric Fournet and Luc Maranget

June 18, 1997

Contents

Foreword

This manual documents the release 1.01 of the join-calculus language; it is organized as follows:

- Part I, "An introduction to the join-calculus", gives an overview of the language in two parts. First, it presents the concurrent language. Then, it introduces the distributed model on a few examples.
- Part II, "User's manual" is the reference description of the join-calculus language and its libraries.
- Part III, "Tools" documents the compiler, the runtime systems, and programming utilities.

Status

This is a first prototype implementation; it does not fully implement the distributed semantics (no remote failure detection). The implementation also has serious problems of efficiency (simple interpreter, no distributed GC).

License

Availability

The complete distributions for both the join-calculus and Objective Caml reside on the machine `ftp.inria.fr`.

The distribution files can be transferred by anonymous FTP:

Host:	`ftp.inria.fr` (Internet address `192.93.2.54`)
Login name:	`anonymous`
Password:	your e-mail address
Directory:	`lang/join/`
Files:	see the index in file `README`

More information is also available on the World Wide Web, `http://pauillac.inria.fr/join/`.

Please report feedback and bugs to `jc-team@inria.fr}`.

Acknowledgments

We thank the Objective Caml development team for their support and advice. We are particularly grateful to Xavier Leroy, who allowed us to freely use much of the source files he wrote.

Part I

An introduction to the Join-Calculus

Chapter 1

Concurrent programming

This part of the manual is a tutorial introduction to the join-calculus. This chapter presents small, local, examples. Chapter 2 deals with the distributed features.

1.1 Conventions

Examples are given as join-calculus source, followed by the output of the compiler when enabled to print types. For instance, here is the definition of a simple top-level value binding:

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

An example may span over several programs chunks. Here is another binding:

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

In the end, the output generated by the execution of the example is shown:

```
# spawn echo(y)
# ;;
⇒ 2
```

Therefore, to execute the example above, it suffices to concatenate the three source chunks above in some file `a.j` (erasing the "`#`" characters), then to compile `a.j` by the command `jcc -i a.j`, and finally to run the produced code by the command `./a.out`.

1.2 Basics

Join-calculus 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. Processes communicate by sending messages on channels or port names. Messages carried by channels are made of zero or more values and channels are values themselves.

By 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 by a single language construct. This feature allows considering (and implementing) channels as functions, when channels are used as functions.

1.2.1 Processes

Processes are the core syntactic class in the join-calculus. The most simple process sends a message on a channel. For instance, the standard library provides the channel `echo`, sending an integer i on `echo` prints i on the console. At top-level, processes are introduced by the keyword "`spawn`". Top-level phrases are terminated by an optional `;;`.

```
# spawn echo(1)
# ;;
#
# spawn echo(2)
# ;;
⇒ 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.

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)
# ;;
⇒ 21
```

Composite processes also include conditionals (`if`'s) and local binding (`let...in`'s). Process grouping is done by using curly braces "`{`" and "`}`".

```
# spawn
#   let x = 1 in
#   {let y = x+1 in echo(y) | echo(y+1)} | echo(x)
# ;;
⇒ 231
```

Once again, the program above may echo the integers `1`, `2` and `3` in any order.

1.2.2 Expressions

The other important syntactic class of the join-calculus is the class of expressions. By contrast with processes, expressions evaluate to some results.

At top-level, expressions are introduced by the keyword "`do`". Expressions also occur in the right-hand side of value bindings or as arguments to message sending. Apart from variables and constants, such as integers, the most simple expression sends some values on a *synchronous* channel. Synchronous channels return an answer, which is made of zero or more results. For instance, the standard library provides the synchronous channel `print`. When given a message consisting of one integer i, `print` outputs i on the console and returns an empty result, consisting of zero values, to its sender.

```
# let x = 1
# ;;
# do print(x)
# ;;
# do print(x+1)
# ;;
val x : int
⇒ 12
```

In the program above `1`, `x` and `x+1` are expressions whose evaluation returns a single value (here an integer). The expressions `print(x)` and `print(x+1)` return empty results. This makes sense with respect to synchronization: as top-level phrases are evaluated by following textual ordering, the program above first outputs `1` and then outputs `2`.

Composite expressions are made of conditionals, value bindings and sequence. Thus one may write:

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

Sequences may also occur inside processes. The general form of a sequence inside a process is *expression* ; *process*, where *expression* must return an empty result. As *expression* can itself be a sequence, composite sequences appear inside processes naturally. Thus, one may write:

```
# spawn
#   print(1) ; print(2) ; echo(3)
# ;;
⇒ 123
```

A sequence may terminate with the empty process that does nothing and is denoted by "", the empty sequence of characters. Thus, an alternative to the previous example is as follows:

```
# spawn
#   print(1) ; print(2) ; print(3) ;
# ;;
⇒ 123
```

Concrete syntax for processes and expressions are purposely similar, when not the same. A noticeable exception is grouping, which is expressed by curly braces "`{`" and "`}`" in the case of processes and by ordinary parenthesis "`(`" and "`)`" in the case of expressions. Due to parser limitations, grouping is mandatory in the branches of an `if` process, i.e., one must write `if` *expression* `then {` $process_1$ `} else {` $process_2$ `}`.

1.2.3 Channels

Channels or port names are the main values of the join-calculus. The standard library already provides many port names, such as `echo` and `print`.

There are two important categories of port names: asynchronous and synchronous port names. Synchronous names (such as `print`) return values, whereas asynchronous channels (such as `echo`) do not. Therefore, message sending on synchronous channels occur inside expressions, whereas the message sending on asynchronous channels occur inside processes. The type-checker flags an error whenever a channel is used in the wrong context.

```
# do echo(1)
# ;;
File "ex11.j", line 1, characters 3-10:
Type error: This is not an expression with an empty result
jcc: Compilation failed for ex11.j
```

Users can create new channels with a new kind of `let` binding, which is not to be confused with ordinary value 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 message contents are bound to formal parameters. For instance, here is the definition of a stuttering `echo` channel:

```
# let echo_twice(x) = echo(x) | echo(x)
# ;;
val echo_twice : <int>
```

The new channel `echo_twice` has type `<int>`, this means that it accepts messages made of one integer. Sending an integer i on `echo_twice` fires an instance of the guarded process `echo(x) | echo(x)`, where `x` is replaced by i. Such message sending is asynchronous in the sense that it is not possible to know when console output actually takes place.

The definition of a stuttering `print` is as follows:

```
# let print_twice(x) = print(x) ; print(x) ; reply
# ;;
val print_twice : <int> -> <>
```

As exposed by its type, `print_twice` is a synchronous port name that takes one integer as argument and returns an empty result. The type checker infers that `print_twice` is synchronous, thanks to the new `reply` construct whose semantics is to send back some (here zero) values as result. Message sending on `print_twice` is synchronous, in the sense that one knows that console output has occurred at the time when `print_twice` answers.

```
# spawn echo_twice(1)
# ;;
#
# do print_twice(2)
# ;;
#
# spawn echo_twice(3)
# ;;
⇒ 221133
```

Observe that, since processes execute concurrently, `221331` would also be a perfectly valid output. Printing both `2`'s before any `3` is the only constraint here, by the synchronous character of `print_twice`.

Channels are polyadic with respect to both arguments and results (when they exist). For instance, `print` accepts one argument and returns an empty result. Another example is the standard library channel `+`, that accepts two arguments and returns one result. The following channel `f` has arity two, both for argument and result, as shown by its type.

```
# let f(x,y) = reply x+y,y-x
# ;;
val f : <int * int> -> <int * int>
```

The type for `f` clearly shows its arities. Polyadic results are exploited by using polyadic value bindings. For instance the following program should print `3` on the console:

```
# let x,y = f(1,2)
# ;;
#
# do print(x)
# ;;
val x : int
val y : int
⇒ 3
```

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

```
# let fib(n) =
#   if n <= 1 then {reply 1}
#   else {reply fib(n-1) + fib(n-2)}
# ;;
#
# do print(fib(10))
# ;;
val fib : <int> -> <int>
⇒ 89
```

Note that, by contrast with value bindings, channel definitions are always potentially recursive.

Port names are first-class values in the join-calculus. They can be sent as messages on other port names and returned as results. As a result, higher order "functions" can be written.

```
# let twice(f) =
#   let r(x) = reply f(f(x)) in
#   reply r
# ;;
val twice : <<'a> -> <'a>> -> <<'a> -> <'a>>
```

The type for `twice` is polymorphic: it includes a type variable `'a` that can be replaced by any type. Thus `twice` is a synchronous channel that takes a synchronous channel of type `<'a> -> <'a>` as argument and returns one result of the same type.

For instance, `'a` can be the type of integers or the type of strings (`^` is string concatenation):

```
# let succ(x) = reply x+1
# ;;
#
# let double(s) = reply s^s
# ;;
#
# do
#   let f = twice(succ)
#   and g = twice(double) in
#   print(f(0)) ; print_string(g("X"))
# ;;
val succ : <int> -> <int>
val double : <string> -> <string>
⇒ 2XXXX
```

1.2.4 Threads

Threads are not part of the join-calculus language, but they are part of the implementation. Threads are some kind of execution units at the implementation level. Threads are created, may suspend then resume, and finally die. Threads match the intuition of a "direct" causality. For instance, all the printing actions in a sequence are to be executed one after another and clearly belong to an unique thread. By contrast, the parallel composition operator "`|`" separates two threads that execute independently.

More precisely, the following program creates three threads:

```
# spawn
#   {print(1) ; print(2) ; print(3) ;} |
#   {{print(4) ;} |
#     print(5) ; print(6) ;}
# ;;
⇒ 561234
```

The output of the program is a mix of the output of its three threads: `123`, `4` and `56`. An output such as `12`, then `4`, then `356` would reveal that the thread printing `1`, `2` and `3` has been preempted before printing `3`.

Sending messages on channels ultimately fires a new process and should thus create a new thread. However, a new thread is not always created. For instance, compare the following definitions that create infinitely many threads and one, never dying, thread:

```
# let rabbit() = print_string("+") ; rabbit() | rabbit ()
# ;;
```

```
#
# let forever() = print_string("-") ; forever() ; forever() ; reply
# ;;
val rabbit : <>
val forever : <> -> <>
```

Sending a message on asynchronous `rabbit`, fires a small thread that terminates spawning two new instances of `rabbit`, thus creating two other thread. By contrast, sending a message on synchronous `forever` blocks the sender thread until the thread fired by the message completes. The implementation takes advantage of this: it does not fire a new thread and executes the process guarded by `forever()` on the sender thread. Here, message sending results in an ordinary function call.

This implementation behavior is exposed by the following program:

```
# spawn
#   {rabbit()} |
#   {print_newline () ; failwith("over !!!")} |
#   {forever() ;}
# ;;
⇒ -------------------
⇒ jcrun a.out: failwith over !!! (from runtime.ml)
```

The program concurrently starts three threads: first, a `failwith(...)` thread that flushes pending output and kills the system; then, two crazy threads, `rabbit()` and `forever()`.

Considering processes only, the program may terminate after an unspecified number of `-`'s and `+`'s has been printed. Considering threads, more `-`'s than `+`'s should appear on the console, since all `-`'s are printed by the same thread, whereas `+`'s are printed by different threads. All these printing threads compete for scheduling with the lethal thread and the `forever()` thread has to be preempted to stop outputting.

As a conclusion, informal reasoning about threads helps predicting program output, taking the implementation into account. What is predicted is likely output and not correct output.

1.3 Modules

The current implementation of the join-calculus supports a simple module system. This provides a clean access to library routines. Additionally, users may structure their programs in several modules, which can be written and compiled separately.

1.3.1 Names from the library

The standard library provides a lot of pre-defined names. The library is made of modules. Inside programs the name x from the module *mod* is accessed as a qualified name *mod*.x. For instance, the library module `ml` gives access to basic operations from the implementation, including operations on floating point numbers (such as conversion, output functions and the exponentiation `exp`).

```
# let one = ml.float_of_string("1.0")
# ;;
```

```
#
# do ml.print_float(ml.exp(one))
# ;;
val one : ml.float
⇒ 2.71828182846
```

An alternative to qualifying names from modules is “opening” modules. Opened modules are searched by the compiler when it discovers a name that is not bound locally.

```
# open ml
# ;;
#
# let one = float_of_string("1.0")
# ;;
#
# do print_float(exp(one))
# ;;
val one : ml.float
⇒ 2.71828182846
```

The compiler starts with module `sys` opened, so that the primitives `print`, `print_string`, `+`, ... are in fact `sys.print`, `sys.print_string`, `sys.+`, ...

Primitives can be seen as (synchronous) port names in one important aspect: users send messages to them and get results from them. However, there is one important difference: primitives are not first class values. As such, primitives cannot be used on their own, they must be applied to their arguments.

```
# let duplicate_arg(f,x) = reply f(x,x)
# ;;
#
# do print(duplicate_arg(+,2))
# ;;
File "ex25.j", line 4, characters 23-24:
this is not a port name: sys.+
jcc: Compilation failed for ex25.j
```

A simple workaround is to create a port name that simply calls the primitive.

```
# let duplicate_arg(f,x) = reply f(x,x)
# ;;
#
# let plus(x,y) = reply x+y
# ;;
#
# do print(duplicate_arg(plus,2))
# ;;
val duplicate_arg : <<'a * 'a> -> <'b> * 'a> -> <'b>
val plus : <int * int> -> <int>
⇒ 4
```

1.3.2 User defined modules

Modules are not restricted to the standard library. Users can create their own modules, compile them separately and link them together into an executable program. For instance, users may write a module `stutter`, that exports two channels `echo` and `print`. Ideally, users first specify the names exported by module `stutter` and their types, by writing an interface file `stutter.ji`.

```
# val echo : <int>
# val print : <int> -> <>
```

The interface `stutter.ji` is compiled by issuing the command `jcc stutter.ji`. This produces an object interface file `stutter.jio`.

Then, the implementation file `stutter.j` contains the actual definitions for `stutter.echo` and `stutter.print`.

```
# let echo(x) = async.echo(x) | async.echo(x)
#
# let print(x) = sys.print(x) ; sys.print(x) ; reply
```

The implementation file `stutter.j` is is compiled by issuing the command `jcc -c stutter.j` (`-c` is the compile-only, do-not-link option). This produces an object implementation file `stutter.jo`.

Now that the module `stutter` is properly compiled, some other implementation file `user.j` can use it.

```
# do stutter.print(1)
# ;;
#
# spawn stutter.echo(2) | stutter.echo(3)
# ;;
```

The implementation file `user.j` can be compiled into `user.jo` by issuing the command `jcc -c user.j`. An executable `a.out` is produced by the command `jcc stutter.jo user.jo` that links the modules `stutter` and `user` together. Alternatively, `a.out` can be produced in one step by the command `jcc stutter.jo user.j`.

Running `a.out` may produce the following output:

⇒ *112233*

See sections 3.9 and 7.3 for a more thorough description of modules.

1.4 Join-patterns

Join patterns significantly extend port name definitions. 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`:

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

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

```
# spawn fruit("apple") | cake("pie")
# ;;
⇒ apple pie
```

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

Join-definitions, such as the one for `fruit` and `cake`, provide a simple mean to express "internal" choice.

```
# spawn fruit("apple") | fruit("raspberry") | cake("pie") | cake("crumble")
# ;;
⇒ raspberry pie
⇒ apple crumble
```

Two cake names must appear on the console and both possible combinations of different fruits and cake types are correct.

Composite join-definitions specify several synchronization patterns.

```
# let apple() | pie() = print_string("apple pie") ;
# and raspberry() | pie() = print_string("raspberry pie") ;
val apple : <>
val raspberry : <>
val pie : <>
```

Observe that only three names `apple`, `raspberry` and `pie` are defined. Thus, `pie` potentially takes part in two synchronizations.

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

```
# spawn apple() | raspberry() | pie()
# ;;
⇒ apple pie
```

Join-patterns are the key programming paradigm in the join-calculus. They allow the encoding of many data structures, which can be used concurrently. For instance, the following source fragment defines a counter:

```
# let Count(n) | inc() = Count(n+1) | reply to inc
# and Count(n) | get() = Count(n) | reply n to get
# ;;
#
# spawn Count(0)
# ;;
val inc : <> -> <>
val Count : <int>
val get : <> -> <int>
```

There definition above calls for two remarks. First, when a join-pattern defines several names, some of which are synchronous, the `reply` construct must now specify the name to which it replies. Hence the new `reply...to` *name* construct.

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 () ;} | {inc() ;}
# ;;
#
# let wait() =
#   let x = get() in
#   if x < 3 then {wait()} else {
#     print_string("three is enough !!!") ; print_newline() ;
#   }
# ;;
#
# spawn wait()
# ;;
val wait : <>
⇒ three is enough !!!
```

Ensuring the correct counter behavior in the example above requires some programming discipline: only one initial invocation on `Count` has to be made. If there are more than one simultaneous invocations on `Count`, then mutual exclusion is lost. If there is no initial invocation on `Count`, then the counter 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() =
#   let Count(n) | inc0() = Count(n+1) | reply to inc0
#   and Count(n) | get0() = Count(n) | reply n to get0 in
#   Count(0) | reply inc0,get0
# ;;
#
# let inc,get = create_counter()
# ;;
val create_counter : <> -> <<> -> <> * <> -> <int>>
val inc : <> -> <>
val get : <> -> <int>
```

This programming style is reminiscent of "object-oriented" programming: a counter is a thing called an object, it has some internal state (here, `Count` and its argument) and 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 now coexist.

1.5 Control structures

Join-pattern synchronization expresses many programming paradigms, either concurrent or sequential.

1.5.1 Control structures for concurrency

Locks

Join-pattern synchronization can be used to emulate simple locks.

```
# let new_lock() =
#   let Free() | lock() = reply to lock
#   and unlock() = Free() | reply to unlock in
#   Free() | reply lock,unlock
# ;;
val new_lock : <> -> <<> -> <> * <> -> <>>
```

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 got the lock releases it by the synchronous call `unlock` that fires another invocation of `Free`.

To give an example of lock usage, we introduce channels that output their string arguments several times:

```
#
# let double(p) =
#   let r(s) = p(s) ; p(s) ; reply in
#   reply r
# ;;
#
# let print_port(s) = print_string(s) ; reply
# ;;
#
# let print16 = double(double(double(double(print_port))))
# ;;
val double : <<'a> -> <>> -> <<'a> -> <>>
val print_port : <string> -> <>
val print16 : <string> -> <>
```

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

```
# spawn {print16("-") ;} | {print16("+") ;}
⇒ +++++----------------+++++++++
```

As threads execute concurrently, their outputs may mix, depending upon scheduling.

Using a lock to delimit a critical section, `-`'s and `+`'s do not mix anymore.

```
# let lock,unlock = new_lock()
# ;;
#
# spawn
#   {lock() ; print16("-") ; unlock() ;} |
#   {lock() ; print16("+") ; unlock() ;}
# ;;
#
val lock : <> -> <>
val unlock : <> -> <>
⇒ ++++++++++++++++----------------
```

Barriers

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

```
# let join1() | join2() = reply to join1 | reply to join2
# ;;
val join1 : <> -> <>
val join2 : <> -> <>
```

Note that the definition above includes two `reply` constructs.

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

```
# spawn
#   {print_string("(") ; join1() ; print_string("a") ; join1() ;
#    print_string(")") ;} |
#
#   {join2() ; print_string("b") ; join2() ;}
# ;;
⇒ (ba)
```

Bi-directional channels

Bi-directional channels are found in many process calculi, such as the pi-calculus for instance. In the pi-calculus, given a 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`?`$x.P$). All processes can both send and receive on the channels they know. By contrast, join-calculus processes that know a channel can only send messages on it. The receiver process is defined at the same time as the channel and is therefore unique. 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 the join-calculus, since join-definitions are binding constructs.

Bi-directional channels can be defined in the join calculus as follows:

```
# let new_pi_channel() =
#   let send(x) | receive() = reply x to receive in
#   reply send,receive
# ;;
val new_pi_channel : <> -> <<'a> * <> -> <'a>>
```

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. One of these messages is received by making a synchronous call to the other port name `receive`. Now let us "translate" the following pi-calculus process:

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

We get:

```
# spawn
#   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(y) ;}
# ;;
⇒ 4
```

Observe that, by contrast with the join-calculus 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_synch_channel() =
#   let send(x) | receive() = reply x to receive | reply to send in
#   reply send,receive
# ;;
val new_pi_synch_channel : <> -> <<'a> -> <> * <> -> <'a>>
```

1.5.2 Loops

Join-patterns are also useful for expressing various programming control structures. A simple control structure is iteration on some integer interval.

Simple loops

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

```
# let loop(a,x) = if x > 0 then {a() | loop(a,x-1)}
# ;;
#
# let echo_star() = print_string("*") ;
# ;;
```

```
#
# spawn loop(echo_star,5)
# ;;
val loop : <<> * int>
val echo_star : <>
⇒ *****
```

When execution order matters, asynchronous loops do not work. Since iterated actions are performed concurrently. Here, a synchronous loop is preferred.

```
# let loop(a,x) = if x > 0 then {a(x) ; loop(a,x-1) ; reply} else {reply}
# ;;
#
# let print_int(x) = print(x) ; print_string(" ") ; reply
# ;;
#
# do loop(print_int,5)
# ;;
val loop : <<int> -> <> * int> -> <>
val print_int : <int> -> <>
⇒ 5 4 3 2 1
```

When a synchronous loop produces a result, this result can be computed inside `reply` constructs and returned. The following example computes the sum of the squares of the integer between `1` and `32`:

```
# let sum(i0,f) =
#   let iter(i) =
#     if i > 0 then {reply f(i)+iter(i-1)} else {reply 0} in
#   reply iter(i0)
# ;;
#
# let square(x) = reply x*x
# ;;
val sum : <int * <int> -> <int>> -> <int>
val square : <int> -> <int>

# do print(sum(32,square))
# ;;
⇒ 11440
```

Port name definitions such as the one for `iter` above belong to a functional programming style. In particular, the various iterations of the loop body (computing `f(i)` here) never execute concurrently, since iterations are performed one after the other.

However, since integer addition is associative and commutative, the summing order of the various `f(i)` does not matters. Thus, asynchronous iteration can be used here. Then, we get a program with much more opportunities for concurrent execution:

```
# let sum(i0,f) =
#
#   let add(dr) | total(r,i) =
#     let r' = r+dr in
#     if i > 1 then {reply total(r',i-1) to total}
#     else  {reply r' to total} in
#
#   let loop(i) = if i > 0 then {add(f(i)) | loop(i-1)} in
#   loop(i0) | reply total(0,i0)
# ;;
#
# do print(sum(32,square))
# ;;
val sum : <int * <int> -> <int>> -> <int>
⇒ 11440
```

Observe how loop result is accumulated using the synchronous name `total`. The argument to the reply to `sum` consists in one call to `total`. This trick enables the synchronous `sum` to return its result when the asynchronous loop is over. In fact, replying directly to `sum` from inside the sub-definition for `add(dr) | total(r,i)` is rejected by the compiler:

```
# let sum(i0,f) =
#
#   let add(dr) | total(r,i) =
#     let r' = r+dr in
#     if i > 1 then {total(r',i-1)}
#     else  {reply r' to sum} in
#
#   let loop(i) = if i > 0 then {add(f(i)) | loop(i-1)} in
#   loop(i0)
# ;;
#
File "ex49.j", line 6, characters 23-26:
continuation outside of its scope: sum
jcc: Compilation failed for ex49.j
```

The current compiler enforces strong restriction on where to place `reply` constructs, see section 7.4.

Distributed loops

Sharing a loop 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. The agent paradigm then serves to allocate computing resources (see section 2.3.3 in the next chapter).

For instance, here are two agents `square1` and `square2`. The agent `square1` modelizes a fast machine, whereas `square2` modelizes a slow machine by computing squares in a contrived way.

Additionally ,`square1` and `square2` differ marginally by their console output: `square1` outputs a `+`, while `square2` outputs a `*` when it starts and a `-` just before answering.

```
# let square1(i) = print_string("+") ; reply i*i
# ;;
#
# let square2(i) =
#    print_string("*") ;
#    let total(r) | wait() = reply r to wait in
#    let mult(r,j) =
#      if j <= 0 then {total(r)} else {mult(r+i,j-1)} in
#    mult(0,i) |
#    let r =  wait() in
#    print_string("-") ; reply r
# ;;
val square1 : <int> -> <int>
val square2 : <int> -> <int>
```

Sharing a loop between several agents is allocating the iterations to be performed to them. The following channel `make_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 make_sum(i0) =
#
#   let add(dr) | total(i) =
#     if i > 1 then {reply dr+total(i-1) to total}
#     else  {reply dr to total}
#   and  wait() = reply total(i0) to wait in
#
#   let loop(i) | register(f) =
#     if i > 0 then {add(f(i)) | loop(i-1) | register(f)} in
#
#   loop(i0) | reply register,wait
# ;;
val make_sum : <int> -> <<<int> -> <int>> * <> -> <int>>
```

The only difference with the asynchronous `sum` loop from the previous section resides in the replacement of the definition `let loop(i) = ...` by the join-pattern definition `let loop(i) | register(f) = ...`. 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 = make_sum(32)
# ;;
#
# spawn register(square1) | register(square2)
# ;;
#
```

```
# do print(wait())
# ;;
val register : <<int> -> <int>>
val wait : <> -> <int>
⇒ ++++++++++++++++++++++++++++++++----------------11440
```

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(f(i))` jobs are spawned asynchronously, so that the different iterations performed by a given agent are executed concurrently. As a result, the iteration space is partitioned evenly between `square1` and `square2`, as illustrated by the output above. This leads to a poor load balance, since the fast `square1` stands idle at the end of the loop, while the slow `square2` is overwhelmed.

A better solution is for an an agent to execute its share of work in sequence rather than concurrently. This is achieved by the slightly modified definition for `loop(i) | register(f)` below:

```
# let make_sum(i0) =
#
#   let add(dr) | total(i) =
#     if i > 1 then {reply dr+total(i-1) to total}
#     else  {reply dr to total} in
#
#   let  wait() = reply total(i0) in
#
#   let loop(i) | register(f) =
#     if i > 0 then
#       {loop(i-1) |
#       let r = f(i) in
#       add(r) | register(f)} in
#
#   loop(i0) | reply register,wait
# ;;
val make_sum : <int> -> <<<int> -> <int>> * <> -> <int>>
```

In the new definitions, `register(f)` is launched again only once `f(i)` is computed. By contrast, `loop(i-1)` is launched immediately, for another agent to grab the next iteration as soon as possible.

```
# let register,wait = make_sum(32)
# ;;
#
# spawn register(square1) | register(square2)
# ;;
#
# do print(wait())
# ;;
val register : <<int> -> <int>>
val wait : <> -> <int>
```

⇒ *+*+*+*+*+*+*+*+*+*+*+*+*+*+*+-11440*

1.6 Data structures in the join-calculus

1.6.1 Simple data structures

Polymorphic pairs can be encoded quite easily in the join-calculus by taking advantage of port name arity.

```
# let create(x,y) =
#   let get() = reply x,y in
#   reply get
# ;;
val create : <'a * 'b> -> <<> -> <'a * 'b>>
```

As exposed by the type above, a pair is a synchronous port name that takes zero arguments and returns two values. The synchronous name `create` returns such a name when given the values to store in the pair.

The content of a pair can be retrieved by sending a message of arity zero on it:

```
# let fst(p) = let x,y = p() in reply x
# and snd(p) = let x,y = p() in reply y
# ;;
val fst : <<> -> <'a * 'b>> -> <'a>
val snd : <<> -> <'a * 'b>> -> <'b>
```

Pairs can now be used in an "abstract" fashion, by considering only the constructor `create` and the two destructors, `fst` and `snd`:

```
# let p = create(1,"coucou")
# ;;
# let x = fst(p) and s = snd(p)
# ;;
# do print(x+string.length(s))
# ;;
val p : <> -> <int * string>
val x : int
val s : string
```

⇒ *7*

An alternative to encoding data structures in the join-calculus is using data structures from Objective Caml. For instance, the standard library provides a module `pair`:

```
# open pair
# ;;
#
# let p =  create(1,"coucou")
```

```
# ;;
# let x = fst(p) and s = snd(p)
# ;;
# do print(x+string.length(s))
# ;;
val p : (int, string) ml.pair
val x : int
val s : string
⇒ 7
```

The type of pairs is now `('a,'b) ml.pair`, emphasizing the abstract character of pairs. Nothing is known about the concrete structure of pairs, implementation details for pairs being local to the implementation of the standard library module `pair`. This is in sharp contrast with the type of encoded pairs: `<> -> <'a * b'>`, which unveils much of the encoding.

1.6.2 Data structure encoding by object-oriented style

A convenient programming style for expressing data structures in the join-calculus is the so-called object-oriented style. An object has an internal, hidden, state and exports some methods, (here some synchronous names) that operate on its internal state. Consider for instance the following encoding for lists:

```
# let cons (h,t) =
#   let head () = reply h
#   and tail () = reply t
#   and self () = reply false,head,tail in
#   reply self
# and nil () =
#   let  head () = reply failwith ("head of nil")
#   and  tail () = reply failwith ("tail of nil")
#   and self () = reply true,head,tail in
#   reply self
# ;;
val cons : <'a * 'b> -> <<> -> <bool * <> -> <'a> * <> -> <'b>>>
val nil : <> -> <<> -> <bool * <> -> <'a> * <> -> <'b>>>
```

The internal state of a list cell is an emptiness status (`true` or `false`), plus, when appropriate, two subcomponents (`h` and `t`). The emptiness status is exposed directly to the external world. The two methods `head` and `tail` give access to the subcomponents of a non-empty list cell or fail. Observe how the name `self` is introduced to pack methods together in the reply to `cons` and `nil`. Finally, the types for the results of `nil` and `cons` are the same. Hence, both kinds of list cells are values of the same type.

List can now be used by directly retrieving methods:

```
# let list_concat(l) =
#   let n,h,t = l() in
#   if n then {reply ""}
```

```
#   else {
#     reply h()^list_concat(t())
#   }
# ;;
val list_concat :
  <'a> -> <string> where 'a = <> -> <bool * <> -> <string> * <> -> <'a>>
```

The above type is recursive. This reflects the fact that lists are recursive data structures. More precisely, the type of the cons-cell creator `cons` was not recursive. However, the recursive type for lists of strings appears naturally when writing a function that traverses lists.

Finally, the following source fragment shows how to create and use string lists:

```
# let replicate(elem,n) =
#   if n <= 0 then {reply nil()}
#   else {
#    reply cons(elem,replicate(elem,n-1))
#   }
# ;;
#
# do print_string(list_concat(replicate("X",16)))
# ;;
val replicate :
  <'a * int> -> <'b> where 'b = <> -> <bool * <> -> <'a> * <> -> <'b>>
⇒ XXXXXXXXXXXXXXXX
```

1.6.3 Mutable data structures

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) =
#   let S(y) | get() = S(y) | reply y to get
#   and S(y) | put(new_y) = S(new_y) | reply to put in
#   S(y0) | reply get,put
# ;;
val create_ref : <'a> -> <<> -> <'a> * <'a> -> <>>
```

Here, the internal state of a cell is its content, its is stored as a message `y` on the channel `S`. 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 : <> -> <int>
val pi : <int> -> <>
val gs : <> -> <string>
val ps : <string> -> <>
```

1.6.4 Elaborate data structure: a concurrent fifo

We are now ready for a more sophisticated example of data structure encoding in the join-calculus. First we define a new kind of list cells. Such a cell always holds an element (`x` below). When created, it stands in the first position of a list, and a name (see `First` below) reflecting that status is activated. Then, when another element is cons-ed is front of the list, a cell additionally holds a pointer to this previous element (see the pattern `First() | set_prev(prev)` below). In the end, a cell can be destroyed (see `kill` below), it then returns both its content, a pointer to the previous cell (when it exists), and a boolean, which is set to `true`, when the destroyed cell is the only one in the list.

```
# let new_cell (x) =
#   let First() | set_prev(prev) = Inside(prev) | reply to set_prev
#   and Inside(prev) | kill () = reply x,prev,false to kill
#   and First()      | kill () = reply x,self,true to kill
#   and self() = reply set_prev,kill in
#   First() | reply self to new_cell
# ;;
val new_cell :
  <'a> -> <'b> where 'b = <> -> <<'b> -> <> * <> -> <'a * 'b * bool>>
```

A fifo is a data structure that provides two methods `put` and `get`. The method `put` stores its argument in the fifo, while the method `get` retrieves one element from the fifo. The internal state of the fifo is either empty (and then the name `Empty` is activated), or it contains some elements (internally, non-empty fifos have the name `S` activated). The name `S` holds two arguments: `fst` is a pointer to the first cell of the element list, while `lst` is a pointer to the last cell of the element list. Thus, elements stored in the fifo are cons-ed in front of `fst` (see `put` below), while retrieved elements are taken from the end of the element list (see `get` below). There is no `Empty() | get()` pattern below. As a consequence, an attempt to retrieve an element from an empty fifo is not an error: answering to `get` is simply postponed until the fifo fills in.

```
# let fifo() =
#   let Empty() | put(x) =
#     let fst = new_cell(x) in
#     S(fst,fst) | reply to put
#   and S(fst,lst) | put(x) =
#      let new_fst = new_cell(x) in
#      let set,rem = fst() in
#      set(new_fst) ;
#      S(new_fst,lst) |
#      reply to put
#   and S(fst,lst) | get () =
#      let set,rem = lst() in
#      let x,prev,last_cell = rem () in
#      if last_cell then {Empty()} else {S(fst,prev)} |
#      reply x to get in
#   Empty() | reply put,get to fifo
```

```
# ;;
val fifo : <> -> <<'a> -> <> * <> -> <'a>>
```

From the fifo point of view, elements are retrieved in the order they are stored. In a concurrent setting this means that when a thread performs several `get` in a row, the retrieved elements come in an order that is compatible with the order in which another thread feeds the fifo.

```
# spawn
#   let put,get = fifo() in
#  {print(get()) ; print(get()) ; print_newline() ;} |
#  {let x = get () and y = get () in } |
#  {put(1) ; put(2) ; put(3) ; put(4);}
# ;;
⇒ 34
```

Therefore, the program above prints two integers from the set `{1, 2, 3, 4}` in increasing order.

1.7 A word on typing

The join-calculus 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:

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

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")
# ;;
#
# do print(i) ; print_string(s)
# ;;
val i : int
val s : string
⇒ 1coucou
```

In other words, the first occurrence of `id` above has type `<int> -> <int>`, while the second has the type `<string> -> <string>`.

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

```
# let S(x) | get() = S(x) | reply x to get
# and S(x) | set(new_x) = S(new_x) | reply to set
val get : <> -> <'_a>
val S : <'_a>
val set : <'_a> -> <>
```

The type variable "`'_a`" that appears inside the types for `S`, `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 `S` 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`" (`S`, `get` and `set` here) are used consistently.

More specifically, if "`'_a`" is instantiated to some type `int`, by sending the message `0` on `S`. Then, the type for `get` is `<> -> <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:

```
# let S(x) | get() = S(x) | reply x to get
# and S(x) | set(new_x) = S(new_x) | reply to set
# ;;
#
# spawn S(0)
# ;;
#
# let x = get()
# ;;
#
# do print_string(x)
# ;;
val get : <> -> <'_a>
val S : <'_a>
val set : <'_a> -> <>
val x : int
File "ex70.j", line 12, characters 3-15:
Type error (Call): this expression has type
<string> -> <>
but is used with type
<int> -> '_b
jcc: Compilation failed for ex70.j
```

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.

```
# let port(p) | arg(x) = p(x)
val port : <<'_a>>
val arg : <'_a>
```

A workaround is to encapsulate the faulty names into another port name definition that defines only one name. This restores polymorphism.

```
# let make_it() =
#   let port(p) | arg(x) = p(x) in reply port,arg
# ;;
val make_it : <> -> <<<'a>> * <'a>>
```

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

```
# let p1,a1 = make_it()
# ;;
#
# let p2,a2 = make_it()
# ;;
#
# spawn p1(echo) | p2(echo_string) | a1(1) | a2("coucou")
# ;;
val p1 : <<'_a>>
val a1 : <'_a>
val p2 : <<'_b>>
val a2 : <'_b>
⇒ 1coucou
```

It is interesting to notice that invoking `make_it()` twice produces two different sets of `port` and `arg` port names, whose types contain different type variables. Thereby, programmers make explicit the different type instantiations that are performed silently by the compiler in the case of generalized type variables.

Non-generalized type variable sometimes are a source of nuisance, when the compiler unexpectedly rejects programs, see section 7.4 for details.

Chapter 2

Distributed programming

This chapter presents the distributed and mobile features of the join-calculus. The join-calculus language is specifically designed to provide a simple and well-defined model of distributed programming. In the previous chapter, we described the core concurrent language, assuming programs were running on a single machine. However, this needs not be the case. Indeed, the join-calculus is entirely based on asynchronous message-passing, which is the basic operation of most distributed systems.

In this chapter, we describe support for execution on several machines and new primitives to control locality, migration, and failure. To this end, we interleave a description of the model with a series of examples that illustrate the use of these primitives.

2.1 The Distributed Model

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

In the join-calculus, 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 process $P|Q$ on a single machine. In particular, the scope for defined names and values is independent of localization: when a port name is known of some process, it can be used to form messages (either as destination or as content) without knowing whether it is locally- or remotely-defined. So far, locality is transparent, and programs can be written independently of where their chunks will actually be executed.

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 dramatically affected because message-sending over the network takes much longer than local calls; finally, the termination of some underlying runtime will affect all its local processes. For all these reasons, locality is explicitly controlled within the join-calculus; it can be adjusted using migration. In contrast, resources such as definitions and processes are never silently relocated by the system.

The name-server

Because the join-calculus 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.

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 (see 5.8 for reference).

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{ let f(x) = reply x*x in ns.register("square",f); }
# spawn{ let sqr = ns.lookup("square") in print(sqr(2)); }
⇒ 4
```

Of course, this makes sense only when the two processes are running as part of stand-alone programs on different machines, and this assumes that these processes share the same strings and conventions.

Running several programs in concert

The runtimes that participate to a distributed computation are launched as Unix commands, e.g. bytecode executables generated by the compiler. Each runtime is given its own join-calculus program, along with options that describe its distributed behavior.

By default, a runtime is in local mode and makes no attempt to communicate with peers. Otherwise, it can run either as a name server (flag `-s`) or as the client of another name server (flag `-c`). In both cases, the name server is specified by its IP address. When `lookup` or `register` is used for the first time in a client runtime, this runtime attempts to communicate with its server; later on, runtimes directly communicate with one another on demand, and there is no further difference between client and server runtimes.

The following example illustrates this with two machines. Let us assume that we rely on a single machine "`here.inria.fr`" for 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`":

```
# let f(x) =
#   print_string("["^ml.string_of_int(x)^"] ");
#   reply x*x
#
# do ns.register("square",f)
```

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

```
here> jcc p.j -o p.out
here> ./p.out -s
```

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 sqr = ns.lookup("square")
# ;;
# let log(s,x) =
#   print_string("q: "^s^"= "^ml.string_of_int(x)^"\n"); reply
# ;;
# let sum(s,n) = reply (if n = 0 then s else sum(s+sqr(n),n-1))
# ;;
# do log("sqr(3)",sqr(3))
# do log("sum(5)",sum(0,5))
```

On another machine `there.inria,fr`, we compile and run our second program (`q.j`) in client mode, using the address of our name server as parameter:

```
there> jcc  q.j -o q.out
there> ./q.out -c -host here.inria.fr
```

What is the output 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` statements: `f` (aliased as `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:

⇒ *`q: sqr(3)= 9`*
⇒ *`q: sum(5)= 55`*

2.2 Locations and Mobility

So far, the localization of processes and expressions is entirely static. In some cases, however, we would need a finer control. To compute the sum of squares in the previous example, each call to `sqr` within the loop resulted in two messages on the network (one for the request, and another one for the answer). It would be better to run the whole loop on the machine that actually computes squares. Yet, we would prefer not to modify the program running on the server every time we need to run a different kind of loop that involves numerous squares.

To this end, we introduce a unit of locality called "location".

A location contains a bunch of definitions and running processes “at the same place”. Every location is given a name, and these location names are first-class values. They can be communicated as contents of messages, registered to the name server, ...exactly as port names. They can also be used in primitives that dynamically change the organization of locations.

2.2.1 Basic examples

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

```
# loc this_location
#   with  square(x) = reply x*x
#     and cubic(x)  = reply square(x)*x
#   init
#     print(square(2));
# end
#
# do print(cubic(2))
val this_location : location
val square : <int> -> <int>
val cubic : <int> -> <int>
⇒ 84
```

This declaration binds a location name `this_location`, and two port names `square` and `cubic` whose scope extends within the location and in the following statements. Here, it also starts a process `{print(square(2));}` in its `init` part. This process runs in the location, in parallel with the remaining part of the program. As a result, we can obtain either `84` or `48`.

Distributed computations are organized as trees of nested locations; every definition and every process is permanently attached to the location where it appears in the source program; a process can create new sublocations with an initial content (bindings and processes) and a fresh name. Once created, there is no way to place new bindings and processes in the location from the outside.

For instance, the following program defines three locations such that the locations named `kitchen` and `living_room` are sublocations of `house`:

```
# loc house
#   init
#     loc kitchen
#       with cook() = print_string("cooking...\n"); reply
#       end
#     and living_room
#       with switch() | off() = print_string("music on\n"); on() | reply to switch
#        and switch() | on()  = print_string("music off\n"); off() | reply to switch
#       init off()
#       end
#     in
#     switch(); cook(); switch();
```

```
#   end
val house : location
⇒ music on
⇒ cooking...
⇒ music off
```

2.2.2 Mobile Agents

While processes and definitions are statically attached to their location, locations can move from one place to another. Such migrations are triggered by a process inside of this location. As a result of the migration, the moving location becomes a sublocation of its destination location.

Notice that locations can be used for several purposes: as a destination addresses, as mobile agents, or as a combination of the two.

Let us propose an agent-based variant of the previous example. On the square side, we create a new empty location "`here`", and we register it on the name-server; its name will be used as the destination address for our mobile agent.

```
# loc here end
# do ns.register("here",here)
```

On the client side, we create another location "`mobile`" that wraps the loop computation that should be executed on the square side; the process within `mobile` first gets the name `here`, then migrates its location inside of "`here`". Once this is done, it performs the actual computation.

```
# loc mobile
#   init
#     let here = ns.lookup("here") in
#     go(here);
#     let sqr = ns.lookup("square") in
#     let sum(s,n) =
#       reply (if n = 0 then s else sum(s+sqr(n),n-1)) in
#     let result = sum(0,5) in
#     print_string("q: sum(5)= "^ml.string_of_int(s)^"\n");
#   end
```

The `go(here)` primitive migrates the whole `mobile` location on machine `here.inria.fr`, as a sub-location of location `here`, then it returns. Afterwards, the whole computation (calls to the name server, to `sqr` and to `sum`) is local to `here.inria.fr`. There are only three messages exchanged between the two machines: one for the `lookup`(here) request, one for the answer, and one for the migration.

Let us consider a variant of `mobile` that combines migration and remote communication:

```
# let sqr  = ns.lookup("square")
# let here = ns.lookup("here")
#
# let log(s,x) =
```

```
#   { let r = ml.string_of_int(x) in
#     print_string("agent: "^s^" is "^r^"\n"); }
#
# loc mobile
#   with quadric(x) = reply sqr(sqr(x))
#   and  sum(s,n,f) = reply (if n = 0 then s else sum(s+f(n),n-1,f))
#   init
#     go(here);
#     log("sum ( i^2 , i= 1..10 )",sum(0,10,sqr))
#   end
#
# spawn log("sum ( i^4 , i= 1..10 )",sum(0,10,quadric))
```

As before, the mobile agent contains a process that first controls the migration, then performs some computation. Here, the location `mobile` is also used as a container for the definitions of `quadric` and `sum`. By scoping rules for such bindings, they can be used in the following expressions that remain on machine `there`.

Once the agent arrives on the square machine `here`, the calls to `sum` and `quadric` become remote calls, as if both functions had been defined and registered on machine `here`. Conversely, messages sent to `log` from the body of the mobile agent arrive on machine `there` where the result is printed.

As we run this program on machine `there`, we obtain the output:

```
⇒ agent: sum ( i^2 , i= 1..10 ) is 385
⇒ agent: sum ( i^4 , i= 1..10 ) is 25333
```

As regards locality, every repeated use of `sqr` is now performed on machine `here`. In the example, the computation of the two sums is entirely local once the agent arrives on machine `here` (one network datagram), which is much more efficient than the equivalent RPC-based program (that would send over sixty network datagrams).

Remember that localization and scopes are independent in the join-calculus: an agent can perform exactly the same actions no matter where it is actually positioned in the location tree. If we forget about the difference between what is local and what is remote, our program produces the same result as a plain program where the locations boundaries and the migration have been removed:

```
# let sqr  = ns.lookup("square")
#
# let log(s,x) = { print_string(s); print(x); print_newline(); }
#
# let quadric(x) = reply sqr(sqr(x))
# and sum(s,n,f) = reply (if n = 0 then s else sum(s+f(n),n-1,f))
#
# spawn log("agent: sum ( i^2 , i= 1..10 ) is ",sum(0,10,sqr))
# spawn log("agent: sum ( i^4 , i= 1..10 ) is ",sum(0,10,quadric))
```

Apart from the performances, both styles are equivalent. In particular, we can first write and test programs, then refine them to get a better tuning of locality.

Applets

The next example shows how to define "applets". An applet is a program that is downloaded from a remote server, then used locally. As compared to the previous examples, this migration operates the other way round. Here, the applet defines a reference cell with destructive reading:

```
# let cell(there) =
#
#   let log(s) = print_string("cell "^s^"\n"); reply in
#
#   loc applet
#     with get() | some(x) = log("is empty"); none() | reply x to get
#     and  put(x) | none() = log("contains "^x); some(x) | reply to put
#     init
#       go(there); none ()
#     end in
#
#   reply get,put
# ;;
# do ns.register ("cell",cell)
```

Our applet has two states: either `none()` or `some(s)` where s is a string, and two methods `get` and `put`. Each time `cell` is called, it creates a new applet in its own location. Thus, numerous independent cells can be created and shipped to callers.

`cell` takes as argument the location (`there`) where the new cell should reside. This is achieved by the process `go(there);none()` that first performs the migration, then sends an internal message to activate the cell. Besides, `cell` defines a `log` function outside of the applet. The latter therefore remains on the server and, when called from within the applet on the client machine, keeps track of the usage of its cell. This is in contrast with applets à la Java: the location migrates with its code, but also with its communication capabilities unaffected.

We complement our example with a simplistic user that allocates and uses a local cell:

```
# let cell = ns.lookup ("cell")
#
# loc user
#   init
#     let get,put = cell(user) in
#     put("world");
#     put("hello, "^get());
#     print_string(get());
#   end
```

⇒ *`hello, world`*

On the server side, we get the trace:

⇒ *cell contains world*
⇒ *cell is empty*
⇒ *cell contains hello, world*
⇒ *cell is empty*

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

Data-driven Migration

In the following examples, we consider large data structures that are distributed among several machines. Because of their relative sizes, it is better to have agents move from site to site as they use the data, rather than move the data or, even worse, access the data one bit at a time. Here, we are interested in defining a general iterator that takes a distributed data structure and applies a function to each of its basic component.

In practice, we use arrays as the building blocs of our data structure; the basic functions to allocate arrays and fill them (`make`), and to create a general function that applies a function over every value in the array (`iter`) could be defined by the following module `table.j`:

```
# let make(n,f) =
#   let a = array.create(n,0) in
#   let loop(m) = if m<n then ( array.set(a,m,f(m)); loop(m+1) ); reply in
#   loop(0); reply a
# ;;
# let iter(a) =
#   let l = array.length(a) in
#   let loop(f,n) = if n < l then ( f(array.get(a,n)); loop(f,n+1) ); reply in
#   let i(f) = loop(f,0); reply in
#   reply i
# ;;
```

We now need to "glue" together several arrays. More precisely, we define an iterator that is consistent with locality: for each array, we move an agent that contains the function to apply inside of the array location, then we apply it, then we move to the next array, ...

Now, each part of the data structure is a pair (host location, iterator), and the mobility protocol consists in (1) migrate the function to apply inside of the host location, (2) call the iterator with this function as argument.

Here is a simple example module (`statistics.j`) that collects data using this protocol and keeps its partial results as an internal message.

```
# let collect(loc_data,iter_data) =
#
#   loc agent
#     with state(n,s,s2) | f(x) = state(n+1,s+x,s2+x*x) | reply to f
```

```
#     and  state(n,s,s2) | done() | result() = reply n,s,s2 to result
#   init
#     go(loc_data);
#     { state(0,0,0) | iter_data(f);done() }
#   end in
#
#   let n,s,s2 = result() in
#   print_string(  "the size is "); print(n);
#   print_string(", the average is "); print(s/n);
#   print_string(", the variance is "); print((n*s2-s*s)/(n*n));
#   print_newline();
```

Here is the definition of a basic data structure that consists of one array:

```
# loc here end
#
# let iter =
#   let f(x) = reply 2*x+5 in
#   table.iter(table.make(100,f))
#
# do ns.register("loc_a",here)
# do ns.register("iter_a",iter)
```

In order to build our structure in a compositional way, we use a `merge` function. This function takes two (location, iterator) pairs and returns a new such pair standing for the compound data structure (module `merge.j`):

```
# let merge(loc1,iter1,loc2,iter2) =
#   loc mobile
#   with iter(f) = go(loc1); iter1(f); go(loc2); iter2(f); reply
#   end in
#   reply mobile,iter
```

Thereby, we can assemble data by repeatedly calling the merge functions on miscellaneous chunks; this defines a structural tree whose leaves are basic arrays. At the same time, `merge` sets up the locations that are needed to traverse it locally.

While in our example basic arrays are stationary, this is not the case for the location of a compound structure: this location moves to each of its subcomponents in turn before applying its function to them.

For instance, if we consider the data structure built from three arrays in the following program,

```
# let itr_a = ns.lookup("iter_a") let loc_a = ns.lookup("loc_a")
# let itr_b = ns.lookup("iter_b") let loc_b = ns.lookup("loc_b")
# let itr_c = ns.lookup("iter_c") let loc_c = ns.lookup("loc_c")
#
# open merge
```

```
# let loc_ab,  itr_ab   = merge(loc_a ,itr_a ,loc_b, itr_b)
# let loc_ab_c,itr_ab_c = merge(loc_ab,itr_ab,loc_c, itr_c)
#
# spawn statistics.collect(loc_ab_c, itr_ab_c)
```

we get some result on the machine that runs the program,

⇒ *the size is 230, the average is 77, the variance is 3109*

and the successive nestings of locations during the computation are:

```
 iter on a   ---->  iter on b  ---->  iter on c
---------------------------------------------------------
 a     b     c   |  a     b     c   |  a     b     c
 ab -->          |       ab         |       ab
 ab_c            |      ab_c -->    |             ab_c
 agent           |      agent       |             agent
```

Notice that migrations are delegated at each level of nesting: as we apply a function `f`, we put it in its own location (here, `agent`) and migrate it inside the data structure location (`ab_c`), which is a leaf of the location tree. Then, we repeatedly apply the function and migrate some location whose branch contains the location of `f`. At any stage, the branch of the location tree that contains the leaf `agent` is the inverse of a branch in the structural tree, and the subpart locations are the successive superlocations of the compound location.

2.3 Termination, Failures and Failure Recovery

As a matter of fact, parts of a distributed computation may fail (e.g., because of the physical crash of a machine). The simplest solution would be to abort the whole computation whenever this is detected, but this is not realistic in case numerous machines are involved. Rather, we would like our programs to detect such failures and take adequate measures (cleanly report the problem, abort related parts of the computation, make another attempt on a different machine, ...).

Accordingly, the join-calculus provides an abstract model of failure and failure detection. This is reflected in terms of locations as follows:

- a location can issue a primitive process `halt()` that, when executed, atomically halts every process inside of this location (and recursively every sublocation);
- a location can detect that another location has halted, using a primitive call `fail(there);P`, such that when P is executed, location `there` is guaranteed to have halted.

The semantics of the join-calculus guarantees that this is the only reason why parts of the computation may stop. Now, an executable running the program `P` on a fallible machine can be thought as a system-defined location `loc machine init P | crash();halt() end` where `crash` may return at any time.

In our model, a machine can in particular remotely detect that another machine has stopped, once it knows the name of a location there. In practice, it is difficult to provide reliable failure detection, as this requires further assumptions on the network.

BEWARE: As it stands, this prototype implementation does not detect this kind of failures: when a runtime terminates abnormally, the failure of its locations is not always detected.

Because locations may only fail as a whole, the programmer can define what are the suitable units of failure, and even use the `halt/fail` primitives to control its computation. Notice that no silent recovery mechanism is provided; the programmer must figure out what to do in case of problems.

2.3.1 Basic examples

To begin with, we use simple examples that attempt to use a port name `say` inside of a fallible location to get messages printed. Because these calls may never return in case the locations stopped, we spawn them instead of waiting for their completion.

In this first example, location `agent` can stop at any time. After the failure occurred, we print some report message. We know for sure that the first `say` can only print something before the failure report, and that the second `say` cannot print anything.

```
# loc agent
#   with say(s) = print_string(s); reply
#   init halt()
#   end
#
# spawn { say("it may work before.\n"); }
#
# spawn { fail(agent);
#         print_string("the location stopped\n");
#         say("it never works after\n"); }
val agent : location
val say : <string> -> <>
⇒ it may work before.
⇒ the location stopped
```

The following example is more tricky. First, the agent does not `halt` itself; however, it migrates within a location that stops and this is a deadly move. Second, the `halt()` process can be triggered only from the outside by a normal message `kill()`. Thus we know that the first `say` always prints its message. Finally, as there is no `halt` in location `agent`, it can only stop because location `failing` halted, so that `fail(agent);` implies `failing` also stopped.

```
# loc failing
#   with kill() = halt()
#   end
#
# loc agent
#   with say(s) = print_string(s); reply
#   init go(failing);
```

```
#   end
#
# spawn { say("it always works.\n"); kill() }
#
# spawn { say("it may work before.\n"); }
#
# spawn { fail(agent);
#         print_string("both locations stopped.\n");
#         say("it never works after.\n"); }
val failing : location
val kill : <>
val agent : location
val say : <string> -> <>
⇒ it always works.
⇒ it may work before.
⇒ both locations stopped.
```

2.3.2 Watching for Failures

We now move on to some more realistic use of failure-detection; we first consider a function that encapsulates a session with mobility and potential failures.

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

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

```
# let safe(job,arg,success,failure) =
#
#   loc box
#     with kill()  = halt()
#     and  start() = reply job(box,arg)  end in
#
#   let done(x)   | running() = done(x) | kill()
#   and done(x)   | failed()  = success(x)
#   and running() | failed()  = failure() in
#
#   done(start()) | running() | fail(box); failed()
val safe : <<location * 'a> -> <'b> * 'a * <'b> * <>>
```

Our supervising protocol either return a result on `success`, or a signal on `failure`. In either case, the message guarantees that no alien computation may take place afterward.

Initially, there is a message `running()`, and the control definition waits for either some result on `done(x)` or some failure detection on `failed()`. Whatever its definition is, the `job` process can create and move locations inside of the box, and eventually return some value to the `start` process within the box. Once this occurs, `done` forwards the reply to the control process, and the first join-pattern is triggered. In this case, the `running()` message is consumed and eventually replaced by a `failed()` message (once the `kill()` message is handled, the box gets closed, and the fail guard in the control process is triggered, releasing a message on `failed`).

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

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

2.3.3 Recovering from partial failures

We finally come back to distributed loops. We give an example of a program that uses the CPU of whatever machine is available to compute the sum $1+2+\ldots+999$. Basically, we only assume that the iteration could be computed in any order, we cut the loop in small chunks, and we distribute a chunk to every available machine. The program takes care of potential failure. In that case, their chunk is aborted and given back to another machine.

```
# let size = 1000
# let chunk = 200
#
# let join(name,there) =
#   loc mobile
#   init
#     let start(i,done) =
#       let loop(u,s) = if u<(i+1)*chunk then { loop(u+1,s+u) } else { done(s) } in
#       loop(i*chunk,0) in
#     go(there);
#     worker(name,mobile,start)
#   end in
#   print_string(name^" joins the party\n");
#
# and job(i) | worker(name,there,start) =
#   print_string(name^","^ml.string_of_int(i*chunk)^"\n");
#   let once() | done(s)  = add(s) | worker(name,there,start)
#   and once() | failed() = print_string(name^" went down\n"); job(i) in
#   once() | start(i,done) | fail(there);failed()
#
# and result(n,s) | add(ds) =
#   let s' = s + ds in
#   if n > 0 then { result(n-1,s') }
```

```
#   else { print_string("The sum is "^ml.string_of_int(s')^"\n"); }
#
# do ns.register("join",join)
#
# spawn
#   result(size/chunk-1,0)
# | let jobs(n) = job(n) | if n>0 then { jobs(n-1) } in jobs(size/chunk-1)
```

The actual work is performed in the `mobile` locations, once they reach the locations `there` provided by joining machines. Messages `job(i)` partition the work to be done. Each remote computation concludes either with a `done(s)` or `failed()` message; in the latter case, the aborted job is re-issued. The resulting sum is accumulated as a message on `result`.

The client is not specific to our computation at all; indeed, its only contribution is a location where others may place their sublocations.

```
# loc worker init
#   let join =  ns.lookup("join") in
#   join("reliable",worker)
# end
```

In the following, we explicitly model an unreliable task force, as a variant of the previous program that also has a "time bomb" which eventually stops the joining location:

```
# let delay = ml.int_of_string(mlsys.getenv("DELAY"))
# let name  = mlsys.getenv("NAME")
#
# loc unreliable_worker init
#   let join =  ns.lookup("join") in
#   let tictac(n) = if n = 0 then { halt() } else { tictac(n-1) } in
#   join(name,unreliable_worker) | tictac(delay)
# end
```

We start three of them with various parameters, for instance with the command lines:

```
DELAY=300  ; NAME=fallible; ./work.out -c &
DELAY=500  ; NAME=dubious ; ./work.out -c &
DELAY=30000; NAME=reliable; ./work.out -c &
./distributed_loop.out -s
```

and we look at the output of the last command (the main loop program):

```
⇒ fallible joins the party
⇒ fallible,0
⇒ dubious joins the party
⇒ dubious,200
⇒ fallible,400
⇒ reliable joins the party
```

```
⇒ fallible went down
⇒ reliable,400
⇒ dubious,600
⇒ reliable,800
⇒ dubious went down
⇒ reliable,600
⇒ The sum is 499500
```

Part II

User Manual

Chapter 3

The Join-Calculus language

Foreword

This document is intended as a reference manual for the Join-Calculus language. It lists the language constructs, and gives their precise syntax and informal semantics. It is by no means a tutorial introduction to the language: there is not a single example. A good working knowledge of the join-calculus 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 [...] denote optional components. Curly brackets {...} 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 (...) denote grouping.

3.1 Lexical conventions

Blanks

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage return, line feed and form feed. Blanks are ignored, but they separate adjacent identifiers, literals and keywords that would otherwise be confused as one single identifier, literal or keyword.

Comments

Comments are introduced by the two characters `(*`, with no intervening blanks, and terminated by the characters `*)`, with no intervening blanks. Comments are treated as blank characters. Comments do not occur inside string or character literals. Nested comments are handled correctly.

Identifiers

$$
\begin{array}{rcl}
\textit{ident} & ::= & \textit{letter}\ \{\textit{letter} \mid \texttt{0}\ldots\texttt{9} \mid \texttt{\_} \mid \texttt{'}\} \\
\textit{letter} & ::= & \texttt{A}\ldots\texttt{Z} \mid \texttt{a}\ldots\texttt{z}
\end{array}
$$

Identifiers are sequences of letters, digits, `_` (the underscore character), and `'` (the single quote), starting with a letter. Letters contain at least the 52 lowercase and uppercase letters from the ASCII set. The current implementation also recognizes as letters all accented characters from the ISO 8859-1 ("ISO Latin 1") set. All characters in an identifier are meaningful. The current implementation places no limits on the number of characters of an identifier.

Integer literals

$$
\begin{array}{rcl}
\textit{integer-literal} & ::= & [\texttt{-}]\ \{\texttt{0}\ldots\texttt{9}\}^{+} \\
& \mid & [\texttt{-}]\ (\texttt{0x} \mid \texttt{0X})\ \{\texttt{0}\ldots\texttt{9} \mid \texttt{A}\ldots\texttt{F} \mid \texttt{a}\ldots\texttt{f}\}^{+} \\
& \mid & [\texttt{-}]\ (\texttt{0o} \mid \texttt{0O})\ \{\texttt{0}\ldots\texttt{7}\}^{+} \\
& \mid & [\texttt{-}]\ (\texttt{0b} \mid \texttt{0B})\ \{\texttt{0}\ldots\texttt{1}\}^{+}
\end{array}
$$

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign. By default, integer literals are in decimal (radix 10). The following prefixes select a different radix:

Prefix	Radix
`0x`, `0X`	hexadecimal (radix 16)
`0o`, `0O`	octal (radix 8)
`0b`, `0B`	binary (radix 2)

(The initial `0` is the digit zero; the `O` for octal is the letter O.) The interpretation of integer literals that fall outside the range of representable integer values is undefined.

String literals

$$
\begin{array}{rcl}
\textit{string-literal} & ::= & \texttt{"}\ \{\textit{string-character}\}\ \texttt{"} \\
\textit{string-character} & ::= & \textit{regular-char} \\
& \mid & \texttt{\textbackslash}\ (\texttt{\textbackslash} \mid \texttt{"} \mid \texttt{n} \mid \texttt{t} \mid \texttt{b} \mid \texttt{r}) \\
& \mid & \texttt{\textbackslash}\ (\texttt{0}\ldots\texttt{9})\ (\texttt{0}\ldots\texttt{9})\ (\texttt{0}\ldots\texttt{9})
\end{array}
$$

String literals are delimited by `"` (double quote) characters. The two double quotes enclose a sequence of either characters different from `"` and `\`, or escape sequences from the table below:

Sequence	Character denoted
`\\`	backslash (`\`)
`\"`	double quote (`"`)
`\n`	newline (LF)
`\r`	return (CR)
`\t`	horizontal tabulation (TAB)
`\b`	backspace (BS)
`\`*ddd*	the character with ASCII code *ddd* in decimal

The current implementation places no restrictions on the length of string literals.

Infix symbols

infix-symbol ::= `<` {*operator-char*} | `>` {*operator-char*} | `< >` | *no-inf-no-sup* {*operator-char*}

operator-char ::= `<` | `>` | *no-inf-no-sup*

no-inf-no-sup ::= `!` | `#` | `$` | `%` | `&` | `*` | `+` | `-` | `.` | `/` | `=` | `?` | `@` | `^` | `|` | `~`

Sequences of "operator characters", such as `<=>` or `!!`, are read as a single token from the *infix-symbol* class. These symbols are parsed as infix operators inside expressions, but otherwise behave much as identifiers.

Warning: due to the concrete syntax of types that includes much `<` and `>` in sequence, the only identifiers that contain only `<` and `>` are `<`, `>` and `<>`.

Keywords

The identifiers below are reserved as keywords, and cannot be employed otherwise:

```
and       do        else        end       external
false     if        in          init      let
loc       open      primitive   reply     spawn
then      to        true        type      val
where     with
```

The following character sequences are also keywords:

```
->    .    |
```

Ambiguities

Lexical ambiguities are resolved according to the "longest match" rule: when a character sequence can be decomposed into two tokens in several different ways, the decomposition retained is the one with the longest first token.

3.2 Values

This section describes the kinds of values that are manipulated by join-calculus programs.

Integer numbers

Integer values are integer numbers from -2^{30} to $2^{30}-1$, that is -1073741824 to 1073741823. The implementation may support a wider range of integer values: on 64-bit platforms, the current implementation supports integers ranging from -2^{62} to $2^{62}-1$.

Character strings

String values are finite sequences of characters. The current implementation supports strings containing up to $2^{24}-6$ characters (16777210 characters).

Booleans

Booleans values are denoted by `true` and `false`.

Port names

Port names are the main value of the join-calculus. They can be seen as channels that carry arguments. There are two kinds of port names: asynchronous ones and synchronous ones. Sending messages on an asynchronous port name is the basic instruction of the language. The invocation of a synchronous name is more like a function call: it gives back results and can occur inside expressions.

Locations

Locations provide support for distributed programming in the join-calculus. A location is a complete runtime unit, with both locally defined port names and definitions and locally running processes. Location may be created, die or migrate.

External values

External values are lifted from the Objective Caml language. External values are abstract values at the join-calculus level, thus no special syntax is provided for them. The standard join-calculus distribution provides floating-point numbers and characters as base external values. Objective Caml data structures such as arrays or lists are also available in the libraries. Other externals values can be defined by the programmer.

NB: In contrast with the other kinds of values, the join-calculus only provides partial distributed support for external values as they move from on runtime to another.

- Basic values (floating-point numbers, chars) and structured values (lists, trees,...) can be used provided they do not contain port names.
- Mutable values can be used (arrays), but remote value-passing does not preserve sharing of their state.
- Non-portable values such as Ocaml functions, Ocaml objects or system file descriptors cannot be sent remotely; this causes a runtime error.

3.3 Primitives

This section describes the kinds of primitives that are used in join-calculus programs. As port names, primitives accept arguments, they can be either synchronous or asynchronous (in the former case, they return some results); they use the same syntax. Notice however that primitives are not first class values: they must be applied to their arguments. As a consequence, they cannot be sent in messages.

Primitives can be of two sorts: either part of the join-calculus or externally defined.

3.3.1 core primitives

The join-calculus provides three core primitives to control migration and failures. The semantics of these primitives depends on the location where they appear.

- `go` is a synchronous primitive that takes a location as an argument, migrates the current location inside of this destination location, then returns an empty result to signal the success of the migration.
- `halt` is an asynchronous primitive that causes the current location to halt (every process in it becomes inert).
- `fail` is a synchronous primitive that takes a location as an argument and returns an empty result to signal that this locations is halted.

3.3.2 external primitives

External primitives are function calls lifted from Objective Caml and linked to the join-calculus runtime system. Most of the standard libraries consist of external primitives; programmers can also extend the join-calculus with their own Ocaml-defined primitives.

NB 1 : The join-calculus does not support the concurrent evaluation of external primitives; as a result, the execution deadlocks if the external call does not return.

NB 2 : The join-calculus provides only partial distributed support for externals primitives: executing a primitive always calls the definition that has been linked to the current runtime, if it exists, or otherwise causes a runtime error.

3.4 Names

Identifiers are used to give names to several classes of language objects and refer to these objects by name later:

- names for values and primitives (syntactic class *name*),
- type constructors (*typeconstr-name*),
- module names (*module-name*),

These three name spaces are distinguished by the context.

Naming objects

$$
\begin{array}{rcl}
\textit{name} & ::= & \textit{ident} \\
 & | & \textit{infix-symbol} \\
\textit{typeconstr-name} & ::= & \textit{ident} \\
\textit{module-name} & ::= & \textit{ident}
\end{array}
$$

As shown above, infix symbols can be used as names.

Referring to named objects

full-name	::=	*name*
	|	*module-name* . *name*
full-infix-name	::=	*infix-symbol*
	|	*module-name* . *infix-symbol*
full-typeconstr-name	::=	*typeconstr-name*
	|	*module-name* . *typeconstr-name*

A named object can be referred to either by its name (following the usual static scoping rules for names) or by an access path *module* . *name*, where *module* designates a module and *name* is the name of an object defined in that module.

3.5 Type expressions

typexpr	::=	*type-ord*
	|	*type-rec*
type-ord	::=	' *ident*
	|	(*type-ord*)
	|	*type-tuple*
	|	*type-tuple* `->` *type-tuple*
	|	*full-typeconstr-name*
	|	*type-ord* *full-typeconstr-name*
	|	(*type-ord* {`,` *type-ord*}) *full-typeconstr-name*
type-tuple	::=	`< >`
	|	`<` *type-ord* {`*` *type-ord*} `>`
typrec	::=	*type-ord* `where` *type-defs*
type-defs	::=	' *ident* `=` *typtop* {`and` ' *ident* `=` *typtop*}

The table below shows the relative precedence and associativity of operators and non-closed type constructions. The constructions with higher precedence come first.

Operator	Associativity
Type constructor application	–
`*`	right
`<`	–

Type expressions denote types in compilation unit interfaces. They occur in the type declaration of values.

Type variables

The type expression `'` *ident* stands for the type variable named *ident*. In abstract data type declarations, type variables are names for the data type parameters. In value type declarations, they represent unspecified types that can be instantiated by any type.

Parenthesized types

The type expression `(` *type-ord* `)` denotes the same type as *type-ord*.

Port types

The type expression `<` $\textit{type-ord}_1$ `*` $\textit{type-ord}_2$ `*...*` $\textit{type-ord}_n$ `>` denotes the type of an asynchronous port name carrying n arguments of type $\textit{type-ord}_1$, $\textit{type-ord}_2$,... $\textit{type-ord}_n$.

The type expression

`<` $\textit{type-ord}_1$ `*` $\textit{type-ord}_2$ `*...*` $\textit{type-ord}_n$ `>` `->` `<` $\textit{type-ord}_{n+1}$ `*` $\textit{type-ord}_{n+2}$ `*...*` $\textit{type-ord}_{n+m}$ `>`

denotes the type of an synchronous port name carrying n arguments of types $\textit{type-ord}_1$, $\textit{type-ord}_2$,... $\textit{type-ord}_n$, and giving back m results of types $\textit{type-ord}_{n+1}$, $\textit{type-ord}_{n+2}$,... $\textit{type-ord}_{n+m}$.

Constructed types

Type constructors with no parameter, as in *full-typeconstr-name*, are type expressions.

The type expression *type-ord typeconstr*, where *full-typeconstr-name* is a type constructor with one parameter, denotes the application of the unary type constructor *full-typeconstr-name* to the type *type-ord*.

The type expression `(` $\textit{type-ord}_1$ `,...,` $\textit{type-ord}_n$ `)` *full-typeconstr-name*, where *full-typeconstr-name* is a type constructor with n parameters, denotes the application of the n-ary type constructor *full-typeconstr-name* to the types $\textit{type-ord}_1$ through $\textit{type-ord}_n$. Parenthesis are optional when n equals one.

Recursive types

The type expression *type-ord* `where` `'` $\textit{ident}_1$ `=` $\textit{typtop}_1$ `and...and` `'` $\textit{ident}_n$ `=` $\textit{typtop}_n$ denotes the same type as *type-ord*, and also binds the type variables $\textit{ident}_1$, ..., $\textit{ident}_n$ to the types $\textit{typexpr}_1$, ... $\textit{typexpr}_n$, both in *type-ord* and in $\textit{typexpr}_1$, ... $\textit{typexpr}_n$. When one of the type variables $\textit{ident}_i$ actually occurs inside one of the type expressions $\textit{typexpr}_j$, a recursive type is created.

3.6 Constants

constant ::= *integer-literal*
| *string-literal*
| `true`
| `false`

The syntactic class of constants comprises literals from the three base types (integers, character strings and booleans).

No special syntax is providesd for external constants (such as floating point numbers). Such constants are introduced by explicitely converting constants of the basic types.

3.7 Join-patterns

$$\begin{array}{rcl} \textit{join-pattern} & ::= & \textit{pattern}\ \{\texttt{|}\ \textit{pattern}\} \\ \textit{pattern} & ::= & \textit{name}\ \texttt{(}\ [\textit{name}\ \{\texttt{,}\ \textit{name}\}]\ \texttt{)} \end{array}$$

Join-patterns are first used to define fresh port names. The simple pattern a (x_1 , ... , x_n) is a definition for a channel a of arity n. Sending n values v_1 , ... , v_n on a results in the firing of a "guarded" process, with the variables x_1 , ... , x_n bound to v_1 , ... , v_n. The variables x_1 , ... , x_n are much like formal parameters and their scope restricts to the guarded process. If a is a synchronous name, then an implicit continuation argument is created, which follows the same scope restrictions as other arguments.

Join-patterns also are templates that are the basic synchronization construct in the join-calculus. A join-pattern of the kind a_1 (x_1 , ... , x_m) | ... | a_n (y_1 , ... , y_l) defines the port names a_1, ... a_n. It also expresses a synchronization between these names. When concurrent invocations on the port names a_1, ... a_n are present, the join-pattern is enabled and some action is to be taken. The action is to fire a process that is guarded by the pattern. At that time, the formal parameters x_1 ,... , x_m, ... , y_1 , ... , y_l, are bound to the actual values given as arguments to the names a_1, ... , a_n.

Patterns are *linear*: a variable cannot appear several times in a given pattern. This applies both to port names and formal arguments. At the moment, there are no anonymous formal parameters: formal arguments must all be variables.

3.8 Processes and expressions

process	::=	*full-name* (*expr-list*)
	|	`reply` *expr-list* [`to` *name*]
	|	
	|	*process* `|` *process*
	|	*expr* ; *process*
	|	`if` *expr* `then` `{` *process* `}` [`else` `{` *process* `}`]
	|	*decl* `in` *process*
	|	`{` *process* `}`
expr	::=	*constant*
	|	*full-name*
	|	*full-name* (*expr-list*)
	|	*expr full-infix-name expr*
	|	*expr* ; *expr*
	|	`spawn` `{` *process* `}`
	|	`if` *expr* `then` *expr* [`else` *expr*]
	|	*decl* `in` *expr*
	|	(*expr*)
expr-list	::=	[*expr* {, *expr*}]
decl	::=	`let` *process-decl*
	|	`let` *expr-decl*
	|	`loc` *agent-decl* `end`
process-decl	::=	*join-pattern* = *process* {`and` *join-pattern* = *process*}
expr-decl	::=	*name* {, *name*} = *expr* {`and` *name* {, *name*} = *expr*}
agent-decl	::=	*agent* {`and` *agent*}
agent	::=	*ident* [`with` *process-decl*] [`init` *process*]

The join-calculus possesses two base syntactic classes: processes and expressions. Processes form the skeleton of the join-calculus as a concurrent language, whereas expressions are more oriented toward functional programming. Both processes and expressions have a similar concrete syntax. Generally, they are discriminated by context: expressions are adequate in a context where a result is needed (such as arguments to message sending), whereas processes occur where some actions are to be performed.

A noticeable exception to the uniformity of concrete syntax is grouping, which is performed using curly braces `{...}` in the process case, and parenthesis `(...)` in the expression case. Such groupings can be used to disambiguate between processes and expressions. This trick is mandatory in the branches of the conditional construct `if...then...else...`, where curly braces `{...}` must be used in the process case.

The table below shows the relative precedences and associativity of operators and non-closed constructions. The constructions with higher precedence come first. For infix symbols, we write

"*..." to mean "any symbol starting with *".

Construction or operator	Associativity
`!...#...$...“.”...~...`	left
`^...@...`	right
`*.../...%...`	left
`+...-...`	left
`<...>...=...`	–
`|...&...`	left
`if`	–
`,`	left
`;`	left
`|`	right
`let loc`	–

3.8.1 Basic constructs

Empty process

The empty process "" does nothing. Its presence can be emphasized by using process grouping {}.

Constants

Expressions consisting in a constant evaluate to this constant.

Names as expressions

When occurring alone in an expression context, names evaluate to the value they are bound to. Names bound to primitives are not allowed as expressions, and cause a runtime-error.

Message passing

The construct a ($expr_1$, ... , $expr_n$) denotes sending the results of the evaluations of $expr_1$, ... , $expr_n$ on the port name (or the primitive) a. The expressions $expr_1$, ... , $expr_n$ are evaluated in some unspecified order. In a process context, the name a must be asynchronous, whereas in an expression context, the name a must be synchronous. This constraint is checked when typing programs.

Note that channels are polyadic in their arguments. Polyadicity also occur at the result level: a synchronous channel takes n arguments and gives back m result, where n and m are any integer greater than or equal to zero.

Some synchronous binary channels can be invoked using infix notation: $expr_1$ *full-infix-name* $expr_2$ is equivalent to *full-infix-name* ($expr_1$, $expr_2$).

Continuation invocation

The process `reply` $expr_1$, ... , $expr_n$ `to` *name* executes by sending the message $expr_1$, ..., $expr_n$ on the continuation of *name*. As the continuation of *name* is an implicit argument to *name*, the `reply...to` *name* construct may only occur in a process guarded by a join pattern that defines *name* (see below). In the current implementation, some additional restrictions apply to continuation usage (see section 7.4).

When *name* is the only channel being defined, the `to` *name* part can be omitted.

Channel definitions

New channels are defined by the *process-decl* construct. The simple declaration *join-pattern* `=` *process* is both a definition for some fresh channel names and the specification of a synchronization pattern on these names (see section 3.7). This synchronization guards *process*.

Channel definitions are recursive definitions: if a is a port named defined by *join-pattern* above, then a may occur free inside *process* and its occurrence refers to the channel defined by the declaration *join-pattern* = *process*.

The composite declaration $\textit{join-pattern}_1$ `=` $\textit{process}_1$ `and...and` $\textit{join-pattern}_n$ `=` $\textit{process}_n$ is a definition for all the names that the simple declarations $\textit{join-pattern}_1$ `=` $\textit{process}_1$, ...$\textit{join-pattern}_n$ `=` $\textit{process}_n$ define. The same channel name a may be defined by several simple declarations inside a composite declaration. Then, only one fresh channel a is defined and channel a takes part to several synchronizations.

Value binding

Any value in the language can be bound to a name by using the *expr-decl* construct. Evaluating a declaration x_1 `,...,` x_n `=` *expr* is evaluating *expr*, which must return n values, which are then bound to x_1 , ..., x_n. Basically, *expr* is a message sending construct on a synchronous channel that gives back n results. Such basic n-ary expressions can be combined by using other language constructs such as conditionals or local definitions.

Value bindings are non-recursive: in x_1 `,...,` x_n `=` *expr*, the scope of $x_1, \ldots, x_n$ does not comprise *expr*. Similarily, the scope of the x^i_j in x^1_1 `,...,` x^1_n `=` $\textit{expr}_1$ `and...and` x^m_1 `,...,` x^m_p `=` $\textit{expr}_m$, does not comprise the $\textit{expr}_j$. Such composite value binding are evaluated in some unspecified order and all the names they bind must be different.

Location definitions

The simple location definition `loc` *name* `with` *process-decl* `init` *process* `end` executes by making a new location, which is bound to *name*, whose initial active channels are defined by *process-decl*, and whose initial active process is *process*. Note that both initial active channels definition and initial active process may be omitted.

The composite location definition $\textit{agent}_1$ `and...and` $\textit{agent}_n$ is the simultaneous definition of n locations. Such a declaration is recursive, that is the scope of the location and initial port names defined by $\textit{agent}_1, \ldots \textit{agent}_n$ comprises the whole composite definition.

Local definitions

The *decl* `in` *process* and *decl* `in` *expr* constructs bind values locally both inside processes and expressions.

They are evaluated by first evaluating *decl*, binding some new names, then evaluating *expr* or executing *process* in an environment extended by the new names defined in *decl*. In an expression context, the result of evaluating the *decl* `in` *expr* construct is the result of evaluating *expr*.

3.8.2 Control structures

Sequence

The expression $expr_1$ `;` $expr_2$ evaluates $expr_1$ first, then $expr_2$, and returns the value of $expr_2$. The process $expr_1$ `;` $process_2$ executes $expr_1$ first, then it executes $process_2$. The expression $expr_1$ must return an empty result. In fact, sequence is a shorthand for a zero-ary local value binding `let =` *expr* `in`....

Concurrent execution

The process $process_1$ `|` $process_2$ executes by concurrently executing $process_1$ and $process_2$. The expression `spawn {` *process* `}` launches the concurrent execution of *process* and returns an empty result.

Conditional

The process `if` $expr_1$ `then {` $process_2$ `} else {` $process_3$ `}` executes $process_2$ if $expr_1$ evaluates to the boolean `true`, and $process_3$ if $expr_1$ evaluates to the boolean `false`. The `else` $process_3$ part can be omitted, in which case it defaults to `else {}`.

The expression `if` $expr_1$ `then` $expr_2$ `else` $expr_3$ evaluates to the value of $expr_2$ if $expr_1$ evaluates to the boolean `true`, and to the value of $expr_3$ if $expr_1$ evaluates to the boolean `false`. The `else` $expr_3$ part can be omitted, in which case it defaults to an expression that returns zero result.

3.8.3 Operators

Symbols from the class `infix-symbols` can appear in infix position (between two expressions). Infix symbols do not have a fixed meaning: they are simply interpreted as message sending on the channels corresponding to the symbols.

The table below lists the symbols defined in the initial environment and their initial meaning. (See the description of the standard library module `sys` in chapter 4 for more details). Their meaning may be changed at any time using `let` *infix-op* `(` x_1`,` x_2 `) =`...

Operator	Initial meaning
`+`	Integer addition.
`-`	Integer subtraction.
`*`	Integer multiplication.
`/`	Integer division. Crashes the system if the second argument is zero. The result is unspecified if either argument is negative.
`%`	Integer modulus. Crashes the system if the second argument is zero. The result is unspecified if either argument is negative.
`^`	String concatenation.
`=`	Structural equality test.
`<>`	Structural inequality test.
`<`	Test "less than".
`<=`	Test "less than or equal".
`>`	Test "greater than".
`>=`	Test "greater than or equal"
`||`	Boolean disjunction
`&&`	Boolean conjunction
`not`	Boolean negation

3.9 Compilation units

A join-calculus program is made of compilation units, also called modules. The source for compilation units reside in different files.

A compilation unit *mod* is made of two source files: the interface source (in a *mod*`.j` file) implementation source (in a *mod*`.ji` file). The Interface specifies the objects that the module exports to the external world, while the implementation contains the join-calculus source for them. (See *join-specification* below).

Additionally, interfaces may be used to connect the join-calculus and Objective Caml, by specifying the join-calculus types of external objects defined in Objective Caml. (See *caml-specification* below).

3.9.1 Interfaces

module-type ::= {*join-specification*}
    | {*caml-specification*}

join-specification ::= `val` *name* : *typexpr*
    | `open` *module-name*

caml-specification ::= `external` *name* : *typexpr* `=` *string-literal*
    | `primitive` *name* : *typexpr* `=` *string-literal*
    | `type` [`(` `'` *ident* {`,` `'` *ident*} `)`] *typeconstr-name* [`=` *string-literal*]
    | `open` *module-name*

Value specifications

The specification of a value exported by an interface is written `val` *name* : *typexpr*, where *name* is the name of the value and *typexpr* its expected type.

External specifications

The specification `external` *name* : *typexpr* = *external-declaration*, specifies that *name* may be used in join-calculus programs, with the type *typexpr*. The string *external-declaration* is an Objective Caml expression that is the actual definition of *name*. The exact nature of what is bound to *name* depends on the type *typexpr*. It can either be a primitive, when *typexpr* is synchronous channel type, or some basic or external value otherwise.

The form `primitive` *name* : *typexpr* = *primitive-declaration* is similar, except that *name* is implemented directly by join-calculus bytecode instructions, as specified in *primitive-declaration*. It is only used in the `sys` module from the standard library.

The type specification `type` [`(` `'` *ident* {`,` `'` *ident*} `)`] *typeconstr-name* informs the join-calculus compiler that *typeconstr-name* is a type constructor whose arity is given by the number of type variables in the part `'` *ident* {`,` `'` *ident*} if present, and is zero otherwise. Parenthesis are optional when only one type variable `'` *ident* is present.

The extended type specification `type` [`(` `'` *ident* {`,` `'` *ident*} `)`] *typeconstr-name* = *caml-type* is similar, except that it specifies the Objective Caml type that implements *typeconstr-name*. This extra-information is used to generate automatic type coercions in calls of externals. It is in fact required for external modules written by users.

Opening a module

The expression `open` *module-name* in an interface does not specify any components. It simply affects the parsing of the following items of the interface, allowing components of the module denoted by *module-name* to be referred to by their simple names *name* instead of path accesses *module-path* `.` *name*. The scope of the `open` stops at the end of the interface file.

3.9.2 Implementations

module-expr	::=	{*top-expr* [`;;`]}
top-expr	::=	*decl*
	|	`do` *expr*
	|	`spawn` *process*
	|	`open` *module-name*

Implementations evaluate by executing top-level expressions in the order they are given.

Value definitions

The declaration *decl* defines new values and bind them to new identifiers, in the same way that a *decl* `in`... construct does (see 3.8.1). The scope of the defined names extends to the end of the implementation file. Such top-level bound values are the *components* of the module *mod*. They may or may not be visible from other modules, depending upon the interface file for *mod*. More precisely, a top-level defined variable *name* is exported form *mod* if the interface of *mod* contains a value specification for `val` *name* : *typexpr*. Additionnaly, the exported type *typexpr* must be an instance of the type inferred by the compiler for the actual value that is bound to *name*.

Expression evaluation

The top-level expression `do` *expr* evaluate expression *expr*, which must return an empty result.

Process spawning

The top-level expression `spawn` *process* concurrently launches *process* and returns an empty result. This is equivalent to `do spawn {` *process* `}`.

Opening a module path

The declaration `open` *module-name* in an implementation does not define any components nor perform any bindings. It simply affects the parsing of the following items of the structure, allowing components of the module denoted by *module-name* to be referred to by their simple names *name* instead of path accesses *module-name* . *name*. The scope of the `open` stops at the end of the implementation file.

3.10 Scheduling and Fairness

The compiler and the runtime provide partial guarantee about fairness in their scheduling strategies. As regards processes triggered in independent definitions, the usual fairness properties apply.

On the contrary, the choice among join-patterns in conflict within the same definition is mostly performed once for all at compile-time, which allows the compiler to generate fewer threads for the processes and more efficient representations for the definitions.

3.10.1 Guaranteed fairness properties

A definition is *active* when enough messages are present to match at least one of its join-patterns; because these messages cannot be consumed otherwise, an active definition remains active until one of its patterns is actually triggered.

The current implementation guarantees that:

1. In an active definition, one join-pattern is eventually triggered
2. If there are enough messages to trigger a join-pattern, then one pattern with at least one name in common with the latter is eventually triggered.

Apart from that, the join-calculus programmer should not rely on a particular scheduling strategy, even if it appears to work.

For instance, the implementation guarantees that the following program eventually prints its message:

```
let loop() = loop()
and done() =  { print_string("over !!\n"); }
spawn { loop() | done() }
```

Conversely, the implementation does *not* guarantee that if there are always enough messages to trigger a pattern, then it is eventually triggered; for instance, when in a definition some pattern is included in another, the latter pattern may never be triggered.

Indeed, the following program will probably run forever without printing anything:

```
let loop()          = loop()
and loop() | done() = { print_string("over !!\n"); }
spawn { loop() | done() }
```

In other words, there is no fairness between the matching join-patterns of an active definition.

Also notice that every call to an external primitive defined in Objective Caml should terminate. As there is no mechanism to enforce it, an external call that runs forever can cause the whole program to deadlock.

3.10.2 Distributed computation and fairness

Assuming that a message within a failed location is not considered present anymore, the previous guarantees hold, no matter of the actual localization of processes and definitions.

There are specific guarantees for the migration primitives:

1. If a halt process is running in a location, then its location branch eventually stops.
2. If a go process is running in a location that remains alive, then eventually the migration succeeds (no matter whether its destination is alive or not; in the latter case, the migration succeeds in stopping the moving location).

A full-fledged implementation of failure-detection should also guarantee that:

1. If a fail-guarded process is running in a location that remains alive, and if the watched location has stopped, then eventually the guarded process is triggered.

The current implementation enforces this last condition only as long as none of the involved runtime terminates.

3.10.3 Termination

When running in local mode, a runtime stops as soon as no pattern may be triggered anymore (i.e., no more active definitions); otherwise a runtime never stops by itself.

A standard library is provided to explicitly perform distributed termination when a program knows the computation is over (cf. 5.12).

3.11 Garbage Collection

This implementation relies on the incremental garbage-collector of Objective Caml; it uses weak pointers for locations and definitions. As a result, the GC can deallocate definitions and empty locations that cannot be reached anymore from the running program.

However, it does not support distributed garbage-collection, which prevents any name that has been sent to another runtime from being deallocated. This can cause serious memory leaks in programs that allocate numerous "distributed" values.

Chapter 4

The core library

This chapter describes the basic types and functions of the join-calculus. They are provided by a special module called `sys`. This module is automatically "opened" when a compilation starts. Hence, it is possible to use unqualified identifiers to refer to the functions provided by the `sys` module, without adding a `open sys` directive.

Conventions

The declarations from the signature of the `sys` module are printed one by one in typewriter font, preceded by its nature (`type`, `value`, `external` or `primitive`) and sometimes followed by a short comment.

Most of these declarations are imported from the Objective Caml system, with the same name. Please refer to the Ocaml manual for more detailed documentation about them.

4.1 Module sys: Atomic types, basic operators and primitives.

Integers

```
type int
external + : <int * int> -> <int>
external - : <int * int> -> <int>
external * : <int * int> -> <int>
external / : <int * int> -> <int>
external % : <int * int> -> <int>
```

Strings

```
type string
external ^ : <string * string>  -> <string>
```

Booleans

```
type bool
```

```
external || : <bool * bool > -> <bool>
external && :  <bool * bool > -> <bool>
external not : <bool> -> <bool>
```

Comparisons

```
external = :  <'a * 'a> -> <bool>
external <> :  <'a * 'a> -> <bool>
external < :  <'a * 'a> -> <bool>
external > :  <'a * 'a> -> <bool>
external <= :  <'a * 'a> -> <bool>
external >= :  <'a * 'a> -> <bool>
```

Basic IO

```
external print_string : <string> -> <>
external print : <int> -> <>
external read : <> -> <int>
external print_newline : <> -> <>
external sys_print : <'a> -> <>
```

Locations, Mobility and Failures

```
type location
primitive null : <>
primitive halt : <>
primitive go : <location> -> <>
primitive fail : <location> -> <>
primitive exit : <> -> <>
external failwith : <string> -> 'a
```

Mostly for debugging purposes

```
external dump_locations : <> -> <>
external dump_state : <> -> <>
external dump_sites : <> -> <>
```

Chapter 5

The standard library

This chapter describes the standard library for the Join-Calculus. Except for the `graphics` library (cf. 5.3), the modules from the standard library are linked to every executable generated by the `jcc` command. In case they contain primitives of Objective Caml, these primitives will be locally accessible from any runtime.

Unlike the core library, the modules from the standard library are not automatically "opened" when a compilation starts. Hence it is necessary to use qualified identifiers to refer to the functions provided by these modules, or to add `open` directives.

Conventions

For easy reference, the modules are listed below in alphabetical order of module names. For each module, the declarations from the signature are printed one by one in typewriter font, preceded by its nature (`type`, `value`, or `external`), and sometimes followed by a short comment.

Most of these declarations are imported from modules of the Objective Caml system, with the same name. Please refer to the Ocaml manual for more detailed documentation about them.

Overview

Here is a short listing, by theme, of the standard library modules.

Data structures:

`char`	p. 74	character operations
`string`	p. 81	string operations
`pair`	p. 80	pair operations
`array`	p. 74	array operations [Ocaml]
`list`	p. 76	list operations
`hashtbl`	p. 76	hash tables and hash functions

System libraries

Distributed libraries

Graphics library

5.1 Module array: array operations

Interface to the Ocaml library module Array

```
open ml
external length : <'a array> -> <int>
external get : <'a array * int> -> <'a>
external set : <'a array * int * 'a> -> <>
external create : <int * 'a> -> <'a array>
external create_matrix : <int * int * 'a > -> <'a array array>
external append : <'a array * 'a array> -> <'a array>
external concat : <'a array list> -> <'a array>
external sub : <'a array * int * int>  -> <'a array>
external copy : <'a array> -> <'a array>
external fill : <'a array * int * int * 'a> -> <>
external blit : <'a array * int * 'a array * int * int> -> <>
external of_list : <'a list> -> <'a array>
external to_list : <'a array> -> <'a list>
external iter : <('a,'b) fun * 'a array> -> <>
external map : <('a,'b) fun * 'a array> -> <'b array>
```

5.2 Module char: some operations on characters

Interface to the Ocaml library module Char

```
open ml
external code : < char > -> < int >
external chr : < int > -> < char>
external escaped : < char > -> < string>
```

5.3 Module graphics: interface to X11 graphics

Interface to the Ocaml library module Graphics

```
open ml
type color
type image
type event
type status
external open_graph : < string > -> <>
external close_graph : <> -> <>
external clear_graph : <> -> <>
external size_x : <> -> <int>
external size_y : <> -> <int>
external rgb : <int * int * int> -> < color>
external set_color : < color > -> <>
external black : color
external white : color
external red : color
external green : color
external blue : color
external yellow : color
external cyan : color
external magenta : color
external background : color
external foreground : color
external plot : <int * int> -> <>
external point_color : <int * int> -> < color >
external moveto : <int * int> -> <>
external current_point : <> -> <int*int>
external lineto : <int * int> -> <>
external draw_arc : <int * int * int * int * int * int> -> <>
external draw_ellipse : <int * int * int * int> -> <>
external draw_circle : <int * int * int> -> <>
external set_line_width : <int> -> <>
external draw_char : < char > -> <>
external draw_string : < string > -> <>
external set_font : < string > -> <>
external set_text_size : <int> -> <>
external text_size : < string > -> <int*int>
external fill_rect : <int * int * int * int> -> <>
external fill_arc : <int * int * int * int * int * int> -> <>
external fill_ellipse : <int * int * int * int> -> <>
external fill_circle : <int * int * int> -> <>
external make_image : < color array array > -> <image>
external dump_image : <image> -> < color array array >
```

```
external draw_image : <image * int * int> -> <>
external get_image : <int * int * int * int> -> < image>
external create_image : <int * int> -> <image>
external blit_image : <image * int * int> -> <>
external wait_next_event : < event list > -> < status >
external mouse_pos : <> -> <int* int>
external button_down : <> -> < bool>
external read_key : <> -> < char>
external key_pressed : <> -> < bool>
external sound : <int * int> -> <>
```

5.4 Module hashtbl: extensible hash tables

Interface to the Ocaml library module Hashtbl

```
open ml
type ('a , 'b) t
external create : <int> -> <('a , 'b) t>
external clear : <('a,'b) t> -> <>
external add : <('a,'b) t * 'a * 'b> -> <>
external find : <('a,'b) t * 'a> -> <'b>
external find_all : <('a,'b) t * 'a> -> <'b list>
external remove : <('a,'b) t * 'a> -> <>
```

5.5 Module list: lists

Interface to the Ocaml library module List

```
open ml
external cons : <'a * 'a list> -> <'a list>
external nil : 'a list
external consp : <'a list> -> <bool>
external length : < 'a list > -> < int>
external hd : < 'a list > -> < 'a>
external tl : < 'a list > -> < 'a list>
external nth : < 'a list  *  int > -> < 'a>
external rev : < 'a list > -> < 'a list>
external append :  < 'a list * 'a list> -> < 'a list>
external flatten : < 'a list list > -> < 'a list>
external iter : < ('a,'b) fun  *  'a list > -> <>
external map : <  ('a,'b) fun   *  'a list > -> < 'b list>
external fold_left : < ('a,('b,'a)fun) fun *  'a  *  'b list > -> < 'a>
external fold_right : <('a,('b,'b) fun) fun * 'a list * 'b > -> < 'b>
external mem : < 'a  *  'a list > -> < bool>
external memq : < 'a  *  'a list > -> < bool>
external assoc : < 'a  *  ('a,'b) pair list > -> < 'b>
```

```
external mem_assoc : < 'a  *  ('a,'b) pair list > -> < bool>
external split : < ('a,'b) pair list > -> < 'a list * 'b list>
external combine : < 'a list  *  'b list > -> <  ('a,'b) pair list>
```

5.6 Module ml: basic types and functions from Ocaml pervasives

Interface to the Ocaml library module Pervasives

NB: by contrast with the Objective Caml module "Pervasives", this module is not open by default.

```
type 'a list
type 'a array
type ('a,'b) pair
type ('a,'b) fun
type unit
type char
type float
type in_channel
type out_channel
external compare : <'a * 'a> -> <int>
external min : <'a * 'a> -> <'a>
external max : <'a * 'a> -> <'a>
external not : <bool> -> <bool>
external abs : <int> -> <int>
external +. : <float * float> -> <float>
external -. : <float * float> -> <float>
external /. : <float * float> -> <float>
external *. : <float * float> -> <float>
external ** : <float * float> -> <float>
external exp : <float> -> <float>
external log : <float> -> <float>
external sqrt : <float> -> <float>
external sin : <float> -> <float>
external cos : <float> -> <float>
external tan : <float> -> <float>
external asin : <float> -> <float>
external acos : <float> -> <float>
external atan : <float> -> <float>
external abs_float : <float> -> <float>
external float : <int> -> <float>
external truncate : <float> -> <int>
external string_of_bool : <bool> -> <string>
external string_of_int : <int> -> <string>
```

```
external int_of_string : <string> -> <int>
external string_of_float : <float> -> <string>
external float_of_string : <string> -> <float>
external stdin : in_channel
external stdout : out_channel
external stderr : out_channel
external print_char : <char> -> <>
external print_string : <string> -> <>
external print_int : <int> -> <>
external print_float : <float> -> <>
external print_endline : <string> -> <>
external prerr_char : <char> -> <>
external prerr_string : <string> -> <>
external prerr_int : <int> -> <>
external prerr_float : <float> -> <>
external prerr_endline : <string> -> <>
external prerr_newline : <> -> <>
external read_line : <> -> <string>
external read_int : <> -> <int>
external read_float : <> -> <float>
external open_out : <string> -> <out_channel>
external flush : <out_channel> -> <>
external output_char : <out_channel * char> -> <>
external output_string : <out_channel * string> -> <>
external output : <out_channel * string * int * int> -> <>
external output_byte : <out_channel * int> -> <>
external output_value : <out_channel * 'a> -> <>
external seek_out : <out_channel * int> -> <>
external pos_out : <out_channel> -> <int>
external out_channel_length : <out_channel> -> <int>
external close_out : <out_channel> -> <>
external open_in : <string> -> <in_channel>
external open_in_bin : <string> -> <in_channel>
external input_char : <in_channel> -> <char>
external input_line : <in_channel> -> <string>
external input : <in_channel * string * int * int> -> <int>
external really_input : <in_channel * string * int * int> -> <>
external input_byte : <in_channel> -> <int>
external input_value : <in_channel> -> <'a>
external seek_in : <in_channel * int> -> <>
external pos_in : <in_channel> -> <int>
external in_channel_length : <in_channel> -> <int>
external close_in : <in_channel> -> <>
```

5.7 Module `mlsys`: system library

Interface to the Ocaml library module Sys

```
external file_exists : < string  > -> <bool>
external remove : < string > -> <>
external rename : < string * string> -> <>
external getenv : < string > -> < string >
external command : < string > -> < int >
external chdir : < string > -> <>
external getcwd : <> -> < string >
```

5.8 Module `ns`: global name server

The name server provides a basic mechanism to initiate distributed computation. Due to lexical scoping, programs running on different machines cannot refer to names that are remotely defined, unless these names were sent to them. To bootstrap the process, a global table of publicly-available names is maintained by one runtime running in server-mode.

According to their distributed protocols, programs will usually first register and/or lookup a few values from this table before the actual computation begins. Once this is done, the name server does not need to be used anymore.

The global ordering of register/lookup calls may depend on the way the different runtimes are started. Accordingly, a lookup to a name that is not defined yet in the nameserver blocks until this name gets registered. Assuming that every name is registered at most once, this guarantees that the values obtained from the name-server, if any, are always the same.

Except for the command-line options (cf. 10) and the availability of the nameserver in case of partial failure, there is no visible difference between server- and client- runtimes.

Beware! In this release no typechecking is performed on the values stored in the table; if a value of some type is registered, then read in a context that expects another incompatible type, anything may happen.

```
val register : <string * 'a> -> <>
```

`ns.register(s,v)` registers some arbitrary value `v` under a given name `s`. The key `s` is a string that must not have been previously registered.

```
val lookup : <string> -> <'b>
```

`ns.lookup(s)` returns the value registered under the name `s`, if any. The key `s` is a string that must have been registered by another process; the function blocks if this is not the case. WARNING: no typechecking on the incoming value.

```
val server : bool
```

`true` if the nameserver is running locally, `false` otherwise.

`val address : string`

The host name used by the name server

`val port : int`

The port number used by the name server

`val fork_client : < string >`

`ns.fork_client(cmd)` starts a new runtime by calling `sh` on the command `cmd` with additional options that specify client mode using the IP address and port number of the current name server. This does not affects the current runtime.

Typical examples of `cmd`s are "`mycode.out`" and
"`rsh there.inria.fr ~/program/mine.out`"

5.9 Module `pair`: pairs

```
open ml
external create : <'a * 'b> -> <('a,'b) pair>
```

Creates an ml pair

`external fst :  <('a,'b) pair> -> <'a>`

Retrieves the first component of an ml pair

`external snd :  <('a,'b) pair> -> <'b>`

Retrieves the second component of a ml pair

5.10 Module `random`: random numbers

Interface to the Ocaml library module Random

```
open ml
external seed : < int > -> <>
external int : < int > -> < int>
external float : < float > -> < float>
```

5.11 Module `sendback`: sending messages from Objective Caml

Generates Ocaml functions from names, allowing asychronous callbacks from Ocaml to the join-calculus.

`external wrap: <<'a>> -> < ('a,ml.unit) ml.fun >`

`sendback.wrap (name)` takes as argument an asynchronous monadic port name `name` and returns an Objective Caml function that takes a value and posts it on that name each time it is called from within an external computation.

5.12 Module `shutdown`: distributed termination

This module implements functions that control the termination of a distributed computation: when a program has completed, it is often suitable for all the participating runtimes to terminate in concert. Distributed termination is a difficult problem in general. In this release, it relies on failure detection and requires specific support from the implementation.

The shutdown protocol requires every participating runtime to register beforehand, usually at the beginning of every program; then, any participant may trigger the global termination.

Current limitations: In this release, some runtimes may report errors such as broken connections or even deadlock instead of exiting silently as a result of a shutdown.

```
val ready: <> -> <>
```

`shutdown.ready()` registers the local runtime as a potential participant to a global shutdown that would occur after it returns.

```
val global: <>
```

`shutdown.global()` causes all registered runtimes to exit.

```
val local: <>
```

`shutdown.local()` causes the current runtime to exit at once. The consequences for the other runtimes is not specified.

5.13 Module `string`: operations on strings

Interface to the Ocaml library module String

```
open ml
external length : < string > -> < int >
external get : < string  *  int > -> < char >
external set : < string  *  int  *  char > -> <>
external create : < int > -> < string >
external make : < int  *  char > -> < string>
external copy : < string > -> < string>
external sub : < string  *  int  *  int > -> < string>
external fill : < string  *  int  *  int  *  char > -> <>
external blit : < string  *  int  *  string  *  int  *  int > -> <>
external escaped : < string > -> < string>
external concat : <string * string list> -> <string>
external uppercase : <string> -> <string>
external lowercase : <string> -> <string>
external capitalize : <string> -> <string>
external uncapitalize : <string> -> <string>
```

5.14 Module unix: interface to Unix system calls

Interface to the Ocaml library module Unix

```
open ml
type error
external error_message : < error > -> < string >
external environment : <> -> < string array >
external getpid : <> -> < int >
external getppid : <> -> < int >
type file_descr
external stdin :  file_descr
external stdout :  file_descr
external stderr :  file_descr
type open_flag
external O_RDONLY : open_flag
external O_WRONLY : open_flag
external O_RDWR : open_flag
external O_NONBLOCK : open_flag
external O_APPEND : open_flag
external O_CREAT : open_flag
external O_TRUNC : open_flag
external O_EXCL : open_flag
external openfile : < string  *  open_flag list  *  int > -> < file_descr>
external close : < file_descr > -> <>
external read : < file_descr  *  string  *  int  *  int > -> < int>
external write : < file_descr  *  string  *  int  *  int > -> < int>
external in_channel_of_descr : < file_descr > -> < in_channel >
external out_channel_of_descr : < file_descr > -> < out_channel >
external descr_of_in_channel : < in_channel > -> < file_descr >
external descr_of_out_channel : < out_channel > -> < file_descr >
type seek_command
external SEEK_SET : seek_command
external SEEK_CUR : seek_command
external SEEK_END : seek_command
external lseek : < file_descr  *  int  *  seek_command > -> < int >
external truncate : < string  *  int > -> <>
external ftruncate : < file_descr  *  int > -> <>
external unlink : < string > -> <>
external rename : < string  *  string > -> <>
external link : < string  *  string > -> <>
external chmod : < string  *  int > -> <>
external fchmod : < file_descr  *  int > -> <>
external chown : < string  *  int  *  int > -> <>
external fchown : < file_descr  *  int  *  int > -> <>
external umask : < int > -> < int >
```

```
external dup : < file_descr > -> < file_descr >
external set_nonblock : < file_descr > -> <>
external clear_nonblock : < file_descr > -> <>
external set_close_on_exec : < file_descr > -> <>
external clear_close_on_exec : < file_descr > -> <>
type dir_handle
external mkdir : < string  *  int > -> <>
external rmdir : < string > -> <>
external chdir : < string > -> <>
external getcwd : <> -> < string >
external opendir : < string > -> < dir_handle >
external readdir : < dir_handle > -> < string >
external rewinddir : < dir_handle > -> <>
external closedir : < dir_handle > -> <>
external pipe : <> -> < file_descr * file_descr >
external mkfifo : < string  *  int > -> <>
external symlink : < string  *  string > -> <>
external readlink : < string > -> < string >
external pause : <> -> <>
external time : <> -> < int >
external gettimeofday : <> -> < float >
external sleep : < int > -> <>
external getuid : <> -> < int >
external geteuid : <> -> < int >
external setuid : < int > -> <>
external getgid : <> -> < int >
external getegid : <> -> < int >
external setgid : < int > -> <>
external getgroups : <> -> < int array >
external getlogin : <> -> < string >
external gethostname : <> -> < string >
external open_process_in : < string > -> < in_channel>
external open_process_out : < string > -> < out_channel>
external open_process : < string > -> < in_channel * out_channel>
external close_process_in : < in_channel > -> <>
external close_process_out : < out_channel > -> <>
external close_process : < in_channel * out_channel > -> <>
```

Chapter 6

Interfacing the Join-Calculus with Objective Caml

This chapter describes how to use types and values defined in Objective Caml from the join-calculus and, to some extent, vice-versa.

Objective Caml types and values that can be used inside join-calculus programs are called *externals*.

6.1 Overview and compilation information

To be seen as externals by the compiler, types and values from Objective Caml must appear in the interface of a join-calculus module. Usually, externals specifications are grouped into specific `.ji` interfaces.

6.1.1 Declaring externals

User externals values and types written in Objective Caml are declared in a join-calculus interface file *mod*`.ji`, using the `type` and `external` keywords.

Declaring types

An external type declaration defines a join-calculus type constructor with the same arity as in Objective Caml. Additionally, it may specify the equivalent Objective Caml type constructor and an iterator. Such extra information enables the automatic translation of values between the two languages while using externals. In the following, this translation process is described as "type coercion".

The most simple declaration of an external type concerns null-ary type constructors:

`type` *typeconstr-name* `=` *string-literal*

This defines the type *mod*`.`*typeconstr-name* as a join-calculus version of *Caml-type*. The *string-literal* above is a Caml type constructor. For instance, here is how the type `float` is declared in the standard library module `ml`.

```
type float = "Pervasives.float"
```

This declaration establishes an equivalence between the Caml type `Pervasives.float` and the join-calculus type `ml.float`. Type coercion between values of the two types is performed.

Another type declaration that applies to type constructors with arguments is as follows:

`type ( '` *ident* {`, '` *ident*} `)` *typeconstr-name* `=` *string-literal*

Parenthesis are optional when only one type variable `'` *ident* is present. The *string-literal* component specifies the Caml equivalent of *typeconstr-name*. It may also provide an optional iterator that is used for type coercion. There are two cases, depending on whether the iterator is present or not.

For instance, here is how the type `list` is declared in the standard library module `ml`.

```
type 'a list = "'a list %List.map"
```

This declaration establishes an equivalence between the Caml type constructor `list` and the join-calculus type constructor `ml.list`. Variable "`'a`" represents a join-calculus type on the left-hand side and an equivalent Caml type on the right hand side. More precisely, given a join-calculus type *jc-type* and and an equivalent Caml type *caml-type*, the two types *jc-type* `ml.list` and *caml-type* `list` are equivalent. Automatic type coercion between object of types *jc-type* `ml.list` and *caml-type* `list` is performed by iterating the coercion operators between *jc-type* and *caml-type*, using the list iterator `List.map`.

Another example is the definition of external pairs in the library module `ml`:

```
type ('a,'b) pair = "'a * 'b %(fun f g (x,y) -> f x,g y)"
```

Observe that the iterator takes two coercion operators "`f`" and "`g`" as arguments. Argument order is defined by the type variable order on the left-hand side of the type declaration. For instance, "`f`" applies to values of type "`'a`" and "`g`" to values of type "`'b`".

Yet another example, where no iterator is specified, is the definition of type constructor `t` from the library module `hashtbl`:

```
type ('a , 'b) t = "('a,'b) Hashtbl.t"
```

This declaration defines the type constructors `hashtbl.t` as the join-calculus version of the Objective Caml type constructor `Hashtbl.t`. Since no iterator is specified, type coercion is pruned at the `hashtbl.t` level (see below).

Finally, the `=` *string-literal* component can be omitted in all kinds of external type declarations. This results in an non-coerced type, a technique that is reserved to system programming.

The complete type equivalence between join-calculus types and Objective Caml types is defined as follows, along with type coercions.

1. The basic types `int`, `string`, and `bool` are equivalent to the homonymous Objective Caml basic types. Values of these types are translated using system provided type coercions.

2. A type variable is equivalent to the same type variable. No type coercion is performed.

3. Constructed types are equivalent when the join calculus type constructor is defined by an external type declaration that establishes the equivalence of type constructors from both languages. Additionally, their type arguments must be equivalent. Arguments must match by following the argument correspondence specified using type variables. Individual argument equivalence is checked as follows:

(a) If an iterator is specified, then unrestricted type equivalence applies. Type coercion is performed, using the specified iterator.

(b) Otherwise, type variables arguments are equivalent to themselves. All other types are equivalent to the Objective Caml type `word`, the types of values in the join-calculus runtime. A simple coercion is performed that does not extends to the subcomponents of constructed values of these types.

4. All other join-calculus types, including channel types and non-coerced types, are equivalent to the Objective Caml type `word`. No type coercion is performed.

Observe that, by rules 2. and 4. above, equivalence with the Objective Caml type `word` disables type coercion. This allows direct access to join-calculus values from within Objective Caml programs, a technique that is reserved to system programming.

Declaring values

In some interface file *mod*`.ji`, external values are declared as follows:

`external` *name* `:` *type* `=` *string-literal*

This defines the name *mod*`.`*name* as a join-calculus value or primitive with type *type*. The component *string-literal* is a string that contains an Objective Caml expression *caml-expr* that defines *name*. How this is done depends upon *type*.

If *type* is a synchronous channel type, then *name* is a primitive, that ultimately calls *caml-expr*, which must be a function of the appropriate type. More precisely, let *type* be

$$\texttt{<}\ type_1\ \texttt{*}\ type_2\ \texttt{*}\ldots\texttt{*}\ type_n\ \texttt{>}\ \texttt{->}\ \texttt{<}\ type_{n+1}\ \texttt{*}\ type_{n+2}\ \texttt{*}\ldots\texttt{*}\ type_{n+m}\ \texttt{>}$$

Then *caml-expr* if a function whose type is:

$$c\text{-}type_1\ \texttt{->}\ c\text{-}type_2\ \texttt{->}\ldots\texttt{->}\ c\text{-}type_n\ \texttt{->}\ (\ c\text{-}type_{n+1}\ \texttt{*}\ c\text{-}type_{n+2}\ \texttt{*}\ldots\texttt{*}\ c\text{-}type_{n+m}\ )$$

where $type_i$ and $c\text{-}type_i$ are equivalent. Type coercion of arguments and results is performed, following the rules of previous section.

Otherwise, *name* is a value, which is computed by applying type coercion to *caml-expr*. Of course, the type of *caml-expr* must be equivalent to *type*.

For instance, here is how the `string_of_float` primitive is declared in the standard library module `ml`:

```
external string_of_int : <int> -> <string>  = "Pervasives.string_of_int"
```

Another example is the `find` primitive from the standard library module `hashtbl`:

```
external find : <('a,'b) t * 'a> -> <'b> = "Hashtbl.find"
```

Remember that the definition of the external type constructor `hashtbl.t` does not supply an iterator. Thus, no type coercion is performed on either keys or contents of hash tables. In the case of the polymorphic `find`, this does not hurt: keys and contents, whose types are the variables "`'a`" and "`'b`" would not be converted anyway. Generally speaking, no iterator is required in the definition of an external type constructor when all the primitives that handles external values of this type are polymorphic.

Finally here are the declarations for the external type `in_channel` and for the standard input from the library module `ml`:

```
type in_channel = "Pervasive.in_channel"
external stdin : in_channel = " Pervasives.stdin"
```

From the join-calculus point of view `ml.stdin` is an external value and automatic type coercion applies.

6.1.2 Implementing externals

User primitives are implemented by Objective Caml functions, whereas external values are implemented by Objective Caml values. Thanks to automatic type coercion, these are written in standard Objective Caml code.

A noticeable exception is system code that belongs to the implementation. Such code must know something about the implemention, including type `word`. A purposely minimal set of declarations is given by the Ocaml module `Ext_imports` from the distribution.

6.1.3 Linking Objective Caml code with join-calculus code

The join-calculus runtime system comprises two main parts: the bytecode interpreter and a set of Objective Caml values that implement externals. Some bytecode instructions are provided to access these externals, and are designated by their offset in a table of Objective Caml values (the table of externals).

In the default mode, the join-calculus linker produces bytecode for the standard runtime system, with a standard set of externals. References to externals that are not in this standard set result in the "external mismatch" code loading error.

In the "custom runtime" mode, the join-calculus linker scans compiled interface files given as arguments and determines the set of required externals. Then, it builds a suitable runtime system, by calling the Objective Caml code linker with:

- the table of the required externals,
- a library that provides the bytecode interpreter and the standard externals,
- Objective Caml compiled interfaces and object code files or libraries (`.cmi`, `.cmo`, or `.cma` files in the limited installation; `.cmi`, `.cmx` or `.cmxa` files in the complete installation) mentioned on the command line for the join-calculus linker, that provide implementations for the user's external.

This builds a runtime system with the required externals. The name of this runtime is given to the linker by the `-custom` *runtime* option. The linker will later generate bytecode for this custom runtime system, provided it was given the right `-jcrun` *runtime* option on the command line. The bytecode includes the necessary information to launch the right runtime.

Thus, to link in "custom runtime" mode, execute the `jcc` command twice. First produce the runtime with:

- the `-custom` *runtime* option,
- the names of the join-calculus interfaces (`.ji` or `.jio` files) that specify the required externals.
- the names of the Objective Caml object files or libraries (`.cmx` or `.cmxa`) that implement the required externals.

This should produce a file *runtime*. This file can be renamed or moved, but this will affect the following linking phase. Second, produce the join-calculus executable with:

- the `-jcrun` *runtime* option, where *runtime* is more a command name than a file name: invoking *runtime* should launch the custom runtime;
- the names of the join-calculus object files (`.jo` files) that make your program.

These two steps can be taken together and there are some shortcuts, see chapter 8 for details.

6.2 Sendbacks from Ocaml to the join-calculus

So far, we have described how to call Ocaml functions from the join-calculus. In this section we show how Ocaml functions can send some messages asynchronously. As a matter of fact, it is not possible for Ocaml code to send synchronous messages (i.e., to wait for answers), since the current implementation assumes that all primitives are non-blocking.

Sendback is only possible on unary asynchronous channels, of type `<'a>`, once these channels have been wrapped into an external value that contains an Ocaml function of type `'a -> unit`. This operation is performed by the name `wrap` from the module `sendback` of the standard library.

```
external wrap: <<'a>> -> < ('a,ml.unit) ml.fun > = ...
```

Once wrapped, the channel has to be passed as an argument to the Ocaml primitive that performs the sendback as an ordinary function call.

6.3 Examples

In this section we give two examples of interfacing Objective Caml with the join-calculus. While building custom runtimes the join-calculus driver (`jcc`) calls the Objective Caml compiler (`ocamlopt` or `ocamlc`). When using `jcc` this way, it is useful to pass it the verbose option `-v` to `jcc`, in order to figure out what is happening.

6.3.1 Linking user Ocaml code

Consider the following Ocaml module interface file `complex.mli`, which provides some operations on complex numbers:

```
type t

val unit_root : int -> t
val print : t -> unit
```

```
val mul : t -> t -> t
```

The function `unit_root` creates a complex number that is an nth root of one, whereas `mul` is complex multiplication. Here is an implementation file for the Ocaml module `Complex`:

```
type t = {re:float ; im:float}

let unit_root n =
  let phi = 2.0 *. acos (-1.0) /. (float n) in
  {re=cos phi ; im=sin phi}

let print  {re=r ; im=i} =
  print_string "{x=" ;
  print_float r ;
  print_string " ; y=" ;
  print_float i ;
  print_string "}"

let mul {re=r1 ; im=i1} {re=r2 ; im=i2} =
  {re= (r1 *. r2) -. (i1 *.i2)  ; im= (r1 *. i2) +. (r2 *. i1)}
```

Ocaml files are compiled using the Ocaml native code compiler by `ocamlopt -c complex.mli complex.ml` (this applies to the joins calculus complete installation; in the limited installation, use the bytecode compiler `ocamlc`).

Then, here is the interface file `complex.ji` that provides the join-calculus with a view on the Ocaml module `Complex`

```
type t = "Complex.t"

external unit_root : < int > -> < t> = "Complex.unit_root"
external print : < t > -> <> = "Complex.print"
external mul : < t  *  t > -> < t> = "Complex.mul"
```

This interface file is compiled by the command `jcc complex.ji`. Then, a new join-calculus runtime is built by the command `jcc -custom jcr-c complex.jio complex.cmx`, where `complex.jio` defines users externals and `complex.cmx` contains Ocaml object code that implements them.

Complex numbers can now be used in some user program `user.j`:

```
let print_roots(n) =
  let r = complex.unit_root(n) in

  let next_root(n,c) =
    complex.print(c) ; print_newline() ;
    if n <= 0 then {reply}
    else {
      next_root(n-1,complex.mul(r,c)) ;
```

```
      reply
    } in

  next_root(n-1,r) ;
  reply
;;

do print_roots(3)
;;
```

Program `user.j` is compiled and linked by issuing the command `jcc -jcrun ./jcr-c user.j`. This produces an executable file `a.out`, whose execution yields:

```
{x=-0.5 ; y=0.866025403784}
{x=-0.5 ; y=-0.866025403784}
{x=1 ; y=-6.10622663544e-16}
```

All the compilation commands given above can be merged into a single command:

```
jcc -custom jcr-c complex.ji complex.mli complex.ml -jcrun ./jcr-c user.j
```

However, separating custom-runtime production and join-calculus compilations is a good idea, since the former takes a lot of time and that the latter is performed more than once during program development.

6.3.2 Importing a library from the Ocaml distribution

The Objective Caml distribution includes many libraries. Not all of them are integrated in the join-calculus default runtime. However, this can be done quite easily. It suffices to write the appropriate `.ji` file and then to supply the adequate Ocaml linking options.

Suppose that we want arbitrary precision arithmetics. The Objective Caml library module `Num` provides such operations, so that we do not have to write any Ocaml code.

First, we write a `num.ji` interface file:

```
type t = "Num.num"

external add : <t * t> -> <t> = "Num.add_num"
    ....
```

Then, we produce our custom runtime `myrun` by issuing the command `jcc -custom myrun nums.cmxa -cclib -lnums num.ji`, where `nums.cmxa` is the Ocaml library that implements the Ocaml module `Num`, using C-primitives from the C-library `nums.a`. A properly installed Ocaml compiler knows where to find these libraries, when given the arguments above.

Finally, the join-calculus source distribution contains an unperfect and non-supported tool to translate `.mli` files into `.ji` files. This tool is `mli2ji` and resides in the `tools` directory.

Part III

Tools

Chapter 7

The compiler (jcc), standard usage

This chapter describes the standard usage of the join-calculus compiler `jcc`, which compiles join-calculus source files to bytecode object files and link these object files to produce standalone bytecode executable files. These executable files are then run by the bytecode interpreter `jcrun`.

7.1 Overview of the compiler

The `jcc` command has a command-line interface similar to the one of most C compilers. It accepts several types of arguments:

- Arguments ending in `.ji` are taken to be source files for compilation unit interfaces. Interfaces specify the names exported by compilation units: they declare value names with their types and declare abstract data types. From the file x`.ji`, the `jcc` compiler produces a compiled interface in the file x`.jio`.

- Arguments ending in `.j` are taken to be source files for compilation unit implementations. Implementations provide definitions for the names exported by the unit, and also contain expressions to be evaluated for their side-effects. From the file x`.j`, the `jcc` compiler produces compiled object bytecode in the file x`.jo`.

  If the interface file x`.ji` exists, the implementation x`.j` is checked against the corresponding compiled interface x`.jio`, which is assumed to exist. If no interface x`.ji` is provided, the compilation of x`.j` produces a compiled interface file x`.jio` in addition to the compiled object code file x`.jo`. The file x`.jio` produced corresponds to an interface that exports everything that is defined in the implementation x`.j`.

- Arguments ending in `.jo` are taken to be compiled object bytecode. These files are linked together, along with the object files obtained by compiling `.j` arguments (if any), and the join-calculus standard library, to produce a standalone executable program. The order in which `.jo` and `.j` arguments are presented on the command line is relevant: compilation units are initialized in that order at run-time, and it is a link-time error to use a component of a unit before having initialized it. Hence, a given x`.jo` file must come before all `.jo` files that refer to the unit x.

The output of the linking phase is a file containing compiled bytecode that can be executed by the join-calculus bytecode interpreter: the command named `jcrun`. If `a.out` is the name of the file produced by the linking phase, the command

`jcrun a.out` arg_1 arg_2 ... arg_n

executes the compiled code contained in `a.out`, passing it as arguments the character strings arg_1 to arg_n. (See chapter 10 for more details.)

On most Unix systems, the file produced by the linking phase can be run directly, as in:

`./a.out` arg_1 arg_2 ... arg_n

The produced file has the executable bit set, and it manages to launch the bytecode interpreter by itself.

7.2 Options

The following command-line options are recognized by `jcc`.

`-c` Compile only. Suppress the linking phase of the compilation. Source code files are turned into compiled files, but no executable file is produced. This option is useful to compile modules separately.

`-i` Cause the compiler to print all defined names (with their inferred types) when compiling an implementation (`.j` file). This can be useful to check the types inferred by the compiler. Also, since the output follows the syntax of interfaces, it can help in writing an explicit interface (`.ji` file) for a file: just redirect the standard output of the compiler to a `.ji` file, and edit that file to remove all declarations of UN-exported names.

`-I` *directory*
Add the given directory to the list of directories searched for compiled interface files (`.jio`) and compiled object code files (`.jo`). By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, in the order in which they were given on the command line, but before the standard library directory.

`-o` *exec-file*
Specify the name of the output file produced by `jcc`. When linking occurs, the default output name is `a.out`, in keeping with the Unix tradition. If linking does not take place (see the `-c` option) and that one file x`.j` is given, the default output name is x`.jo`.

`-v` Cause the compiler to print various messages on what it is doing. This flag can be given twice to make the compiler very talkative. This is useful for debugging your programs, your installation or the compiler itself.

`-comment`
Include comment in produced code. This is useful for debugging your programs (see the `-t` trace option in the description of `jcrun` in chapter 10).

`-noanalyze`
: Suppress automata optimization.

`-noschedule`
: Suppress scheduling optimizations.

`-noopt`
: Suppress all optimizations.

The last three options are merely here for compiler development purposes. Enabling them may produce incorrect executables.

7.3 Modules and the file system

This short section is intended to clarify the relationship between the names of the modules corresponding to compilation units and the names of the files that contain their compiled interface and compiled implementation.

The compiler always derives the module name by taking the base name of the source file (`.j` or `.ji` file). That is, it strips the leading directory name, if any, as well as the `.j` or `.ji` suffix. For instance, compiling the file `mylib/misc.j` provides an implementation for the module named `misc`. Other compilation units may refer to components defined in `mylib/misc.j` under the names `misc.`*name*; they can also do `open misc`, then use unqualified names *name*.

The `.jio` and `.jo` files produced by the compiler have the same base name as the source file. Hence, the compiled files always have their base name equal to the name of the module they describe (for `.jio` files) or implement (for `.jo` files).

When the compiler encounters a reference to a free module identifier `mod`, it looks in the search path for a file `mod.jio` and loads the compiled interface contained in that file. As a consequence, renaming `.jio` files is not advised: the name of a `.jio` file must always correspond to the name of the compilation unit it implements. It is admissible to move them to another directory, if their base name is preserved, and the correct `-I` options are given to the compiler.

Compiled bytecode files (`.jo` files), on the other hand, can be freely renamed once created. That's because the linker never attempts to find by itself the `.jo` file that implements a module with a given name: it relies instead on the user providing the list of `.jo` files by hand.

7.4 Common errors

This section describes and explains the most frequently encountered error messages. Apart from lexing and syntax errors you are more likely to encounter the following errors:

`cannot find file` *filename*
: The named file could not be found in the current directory, nor in the directories of the search path. The *filename* is either a compiled interface file (`.jio` file), or a compiled bytecode file (`.jo` file). If *filename* has the format *mod*`.jio`, this means you are trying to compile a file that references identifiers from module *mod*, but you have not yet compiled an interface for module *mod*. Fix: compile *mod*`.ji` or *mod*`.j` first, to create the compiled interface *mod*`.jio`.

If *filename* has the format *mod*`.jo`, this means you are trying to link a bytecode object file that does not exist yet. Fix: compile *mod*`.j` first.

If your program spans several directories, this error can also appear because you haven't specified the directories to look into. Fix: add the correct `-I` options to the command line.

`this expression has type` t_1`, but is used with type` t_2

This is by far the most common type error in programs. Type t_1 is the type inferred for the expression (the part of the program that is displayed in the error message), by looking at the expression itself. Type t_2 is the type expected by the context of the expression; it is deduced by looking at how the value of this expression is used in the rest of the program. If the two types t_1 and t_2 are not compatible, then the error above is produced.

`name` *name* `has no continuation`

This error is flagged by the typechecker when Name *name* is not a synchronous name and that a `reply ... to` *name* construct exist in the program. This may happen using the abbreviated `reply ...` construct, which is parsed as `reply ... to sys.null` and patched later, after typechecking.

name`:` t_1 `is not an instance of` t_2

On the one hand, the name *name* is defined in an implementation (a `.j` file) and the compiler inferred type t_2 for it; on the other hand *name* is made public in the corresponding interface (a `.ji` file) with type t_1. The error occurs when t_1 is not a valid instance (i.e., a more specific type) of t_2.

`required name:` *name* `is  not provided`

This error occurs when an interface exports *name* and that the corresponding compilation unit does not define *name*.

`type of exported` *name* `is not closed:` *t*

This error occurs when *name* is defined in a `.j` file with no `.ji` associated file. Hence, *name* is automatically exported from the module. Type variables (`'a`, `'b`, ...) in a type *t* can be in either of two states: generalized (which means that the type *t* is valid for all possible instantiations of the variables) and not generalized (which means that the type *t* is valid only for one instantiation of the variables. In a `let` binding `let` $name_1$(x) `|` $name_2$(y) `=` *proc*, the type-checker tries to generalize as many type variables as it can. Generalizing type variables that appear in the types of several co-defined names ($name_1$ and $name_2$ here) would lead to runtime error, when these names are used inconsistently. Note that non-generalized type variable are prefixed by an underscore in compiler output (they yield: `'_a`, `'_b`,...). Another way to introduce non-generalized type variable is using the `let` *name* `=` *expr* construct, where type variable are *never* generalized.

Non-generalized type variables in a type cause no difficulties inside compilation unit (the contents of a `.j` file) but they cannot be allowed inside interfaces (`.ji` or `.jio` file), because they could be used inconsistently later. Therefore, the compiler flags an error when a compilation unit defines and exports a name *name* whose type contains non-generalized type variables. There are three ways to fix this error:

- Add a `.ji` file to give a monomorphic type (without type variables) to *name*. For instance, assuming you need a reference cell for holding integers, instead of writing only the `irefcell.j` file

  ```
    let get()  | state(s) = state(s) | reply s to get
    and set(s) | state(t) = state(s) | reply to set
  (*
  jcc -i output :
    val get : <> -> <'_a>
    val set : <'_a> -> <>
    val state : <'_a>
  *)
  ```

  write a `irefcell.ji` file:

  ```
  val get : <> -> <int>
  val set : <int> -> <>
  val state : <int>
  ```

- If the faulty names do not need to be exported, then add a `.ji` file that does not export them.
- If you really need names with polymorphic types, then their definitions should be changed. The most simple way to do this is to encapsulate the faulty names together in a local definition. For instance, to make a polymorphic reference cell, write the following `refcell.j` file:

  ```
  let ref(s0) =
    let get()  | state(s) = state(s) | reply s to get
    and set(s) | state(t) = state(s) | reply to set in
    state(s0) | reply get,set to refcell
  ;;
  ```

  Then both names `get` and `set` are packed together as a result of an invocation of `refcell` and their common type variable is generalized when the type of refcell is generalized.

  ```
  # jcc -i refcell.j -c
  val ref : <'a> -> <<> -> <'a> * <'a> -> <>>
  ```

  To use `get` and `set` from another module, you will have to create fresh reference cells, thereby creating two fresh instances of `get` and `set`:

  ```
  let gi,si = refcell.ref(0)
  ;;
  let gs,ss = refcell.ref("")
  ;;
  ```

  The `jcc -i` output for compiling this file is:

  ```
  val gi : <> -> <int>
  val si : <int> -> <>
  ```

```
val gs : <> -> <string>
val ss : <string> -> <>
```

name `:` t_1 `is not a closed instance of` t_2

This error is similar to the previous one. On the one hand, the name *name* is defined in an implementation (a `.j` file) and the compiler inferred type t_2 for it; on the other hand *name* is made public in the corresponding interface (a `.ji` file) with type t_1. The error occurs when t_1 is not specific enough to instantiate the non-generalized variables of t_2.

`undefined global` *mod.name*

This error appears when trying to link an incomplete or incorrectly ordered set of files. Either you have forgotten to provide an implementation for the compilation unit named *mod* on the command line (typically, the file named *mod*`.jo`) Fix: add the missing `.j` or `.jo` file to the command line. Or, you have provided an implementation for the module named *mod*, but it comes too late on the command line: the implementation of *mod* must come before all bytecode object files that reference *mod*. Fix: change the order of `.j` and `.jo` files on the command line. Note that mutually recursive modules are not possible. The reason for this is that modules not only define name but also include code to be executed to initialize these names. This initialization code must be executed in some order that guarantees that names are initialized before they are used

`unbound var:` *name*

This error occurs when a program (a `.j` file) refers to a name and when the compiler is unable to find a definition for that name. When *name* is a qualified name *mod.name*, the compiler has found a *mod*`.jio` file but this file contains no declaration for *name*. Usually, *name* is misspelled or the *mod*`.jio` file that is read is not the expected one.

`unbound type constr:` *name*

This error occurs when an interface (a `.ji` file) refers to a type constructor *name* (`sys.int`, `ml.hashtbl` are type constructors) and when the compiler is unable to find a definition for that name.

`this is not a port name:` *mod.name*

Primitives such as “`halt`” or externals such as “`print_string`” are not real join-calculus port names. They can be applied to their arguments (as in `print_string("coucou")`) but cannot be passed as arguments or given back as results (as in `reply print_string to ...`). This error occurs in the latter situation, when a primitive or external is used as a first-class name. Fix: introduce a new definition, for instance the following program does not compile and an error `this is not a port name: sys.print_string` is flagged:

```
let call(p,x) = p(x) ; reply
;;

do call(print_string,"coucou")
;;
```

Whereas this one does compile:

```
let call(p,x) = p(x) ; reply
;;

let my_print_string(s) = print_string(s) ; reply
;;

do call(my_print_string,"coucou")
;;
```

`continuation outside of its scope:` *name*

Synchronous names carry an implicit extra continuation argument. The name *name* is determined synchronous when its definition includes `reply ... to` *name* constructs, which are translated by the compiler into invocations on the implicit continuation of *name*. For efficiency reasons, continuation are treated specially. In particular, continuations cannot occur free in a definition, as the continuation of `f` in the following example, where an error `continuation outside of its scope: f` is flagged:

```
let f(x) =
  let g(y) = reply x+y to f in
  f(x+1)
;;
```

However, the error may also stem from a misspelling in a continuation name.

`threading: double answer on the continuation of` *name*

To enable compilation of synchronous calls as function calls as well as a simple threading policy, continuation usage (i.e. `reply ... to ...` constructs) is severely restricted. This error is flagged when two different threads both contain an invocation on the continuation of *name*, as in `reply 1 to` *name* `| reply 2 to` *name*.

`threading: no answer on the continuation of` *name*

This error is flagged when one branch of a thread contains an invocation on the continuation of *name*, whereas another does not, as in `if b then {reply to` *name*`} else {}`.

All other errors are bugs, either in the compiler or documentation. Please report them.

Chapter 8

The compiler (jcc), advanced usage

This chapter describes the advanced usage of the join-calculus compiler. Advanced usage is linking together join-calculus and Objective Caml code, producing both bytecode executable files and "custom" runtimes to execute them.

Custom runtimes come in two flavors. Basically, a join-calculus runtime is a program written in Objective Caml. Thus, it can be compiled into Objective Caml bytecode (by using `ocamlc`) or into native code (by using `ocamlopt`). Bytecode runtimes favor compilation speed, while native code runtime favor execution speed.

The join-calculus system can be either fully installed (when both `ocamlc` and `ocamlopt` exist on your machine) or installed in a limited fashion (when `ocamlopt` is not available). Both flavors of join-calculus runtimes are available only when the join-calculus distribution is fully installed (see the installation instructions). In case of a limited installation, bytecode runtimes are the only option.

8.1 When should I use custom runtimes ?

Whenever you want to link together join-calculus and Caml code that is not available in the standard join-calculus runtime. Additionally, due to a limitation in the Objective Caml distribution, the graphics library is available only to bytecode runtimes. Since the standard join-calculus runtime is a native code runtime, it does not provide graphics. Thus, using graphics requires generating a custom join-calculus runtime (see the `-graphics` option below). This does not apply in the limited installation where graphics are included in the standard runtime.

8.2 Advanced compiler arguments

The join calculus compiler decodes the following additional arguments.

- Arguments ending in `.jio` are compiled interfaces. They contain definitions for externals (i.e., join calculus names that are in fact Objective Caml expressions). These definitions are used for producing custom runtimes with a custom external table that includes library externals and user defined externals.

- Arguments ending in `.ji` are taken to be source files for compilation unit interfaces. These are compiled into a `.jio` file and used for producing custom runtimes.
- Arguments ending in a caml suffix (i.e., `.mli`, `.ml`, `.cmo`, `.cmi`, `.cma`, `.cmx` or `.cmax`) are taken to be caml files. They are passed to the Objective Caml compiler while building custom runtimes.
- Arguments ending in `.c` are passed to the C compiler, which generates a `.o` object file. This object file is linked with the custom runtime. Note that this is very advanced usage.
- Arguments ending in `.o` or `.a` are assumed to be C object files and libraries. They are passed to the Objective Caml linker that passes them to the Unix linker when producing ultra-custom runtimes.

The output of the advanced linking phase is a bytecode executable (let us say `a.out`) and a custom runtime (let us say `myjcrun`).

`myjcrun a.out` arg_1 arg_2 ... arg_n

executes the compiled code contained in `a.out`, passing it as arguments the character strings arg_1 to arg_n.

On most Unix systems, the file produced by the linking phase can be run directly, as in:

`./a.out` arg_1 arg_2 ... arg_n

The produced file has the executable bit set, and it manages to launch the appropriate bytecode interpreter by itself. Some errors occurs at the Unix level when the operating system cannot find `myjcrun`. An error is flagged by `myjcrun` when its external tables does not allow the execution of `a.out`.

8.3 Advanced compiler options

The following advanced command-line options are recognized by `jcc`.

`-custom` *exec-name*
: Generate a custom runtime of the given name. This custom runtime is built from the appropriate join-calculus compiled interface `.jio` files and from Objective Caml or C object files or libraries.

`-jcrun` *exec-name*
: When generating join-calculus bytecode, the linker will use the external table of *exec-name* which must exist and be executable. The join-calculus executable will run on *exec-name* or on any runtime that is compatible with it.

`-native`
: Generate a native code custom runtime. This is the default. This option is not available in the limited installation.

`-bytecode`
: Generate a bytecode custom runtime, this is the default and only possibility in the limited installation.

`-graphics`
: Include the graphics library from Objective Caml in the custom runtime. This option automatically sets the `-bytecode` flag.

`-camlopt` *option*
: Add *option* as an option to the Objective Caml compiler, while building a custom runtime. For instance, `-camlopt "-I` *dir*`"` causes the Objective Caml compiler to add directory *dir* in its search path.

`-cclib -l`*libname*
: Pass the `-l`*libname* option to the C linker when linking in "custom runtime" mode. This causes the given C library to be linked with the program.

`-ccopt` *option*
: Pass the given option to the C compiler and linker, when linking in "custom runtime" mode (see the `-custom` option). For instance, `-ccopt -L`*dir* causes the C linker to search for C libraries in directory *dir*.

8.4 Common errors

Errors while generating custom runtimes may come from the Object Caml compiler, which in turn may report errors from the C compiler. This complicates correcting such errors. A few errors are generated by `jcc`:

`Cannot create directory for temp files`
: The join-calculus compiler cannot create a subdirectory in the `/tmp` to hold its temporary files, while generating a custom runtime. Fix: usually `/tmp` is full, clean it and try again.

`Customize failed`
: The join-calculus compiler cannot execute *runtime-name* to extract the external table from the runtime given by the `-jcrun` *runtime-name* option, while generating custom bytecode to be executed by *runtime-name*. Fix: make sure that *runtime-name* is a valid invocation of an existing runtime. This may require installing *runtime-name* in your execution path or specifying *runtime-name* as an absolute file name.

`Cannot read config file`
: The join-calculus compiler successfully executed *runtime-name* to extract the external table from the runtime given by the `-jcrun` *runtime-name* option, but was then unable to read it. Fix: difficult, something may be wrong in your join-calculus installation or `/tmp` may be full.

`Custom runtime generation failed`
: The invocation of `ocamlc` or `ocamlopt` failed, while generating a custom runtime. Fix: difficult, have a look at what `jcc` did by reissuing your compilation command with the `-v` flag set

and refer to the Objective Caml documentation. Such errors may arise, for instance, when the type of an external, as specified in a `.ji` file, does not correspond with its type, as inferred by the Objective Caml compiler.

`Error in externals: in` *mod*`, cannot find external:` *name*
: The join-calculus linker cannot find the external *name* in module *mod*, while producing custom bytecode. That is, the external *mod*`.`*name* does not exist in the runtime *runtime-name*, as specified by the option `-jcrun` *runtime-name*. Fix: probably, *runtime-name* is not the right runtime, or some `.ji` file was missing or incomplete at the time *runtime-name* was built.

`Error in externals: double definition of external:` *mod*`.`*name*
: The external *mod*`.`*name* is defined more than once in the the custom runtime *runtime-name*, as specified by the option `-jcrun` *runtime-name*. Note that this error is flagged a bit late, while attempting to generate join-calculus executables intended to run on *runtime-name*.

All other errors are bugs, either in the compiler or documentation. Please report them.

Chapter 9

The dependency generator (jcdep)

The `jcdep` command scans a set of source files (`.j` and `.ji` files) for references to external compilation units, and outputs dependency lines in a format suitable for the `make` utility. This ensures that `make` will compile the source files in the correct order, and recompile those files that need to when a source file is modified.

The typical usage is:

`jcdep` *options* `*.ji *.j > .depend`

where `*.ji *.j` expands to all source files in the current directory and `.depend` is the file that should contain the dependencies.

9.1 Options

The following command-line option is recognized by `jcdep`.

`-I` *directory*
: Add the given directory to the list of directories searched for source files. If a source file `foo.j` mentions an external compilation unit `bar`, a dependency on that unit's interface `bar.jio` is generated only if the source for `bar` is found in the current directory or in one of the directories specified with `-I`. Otherwise, `bar` is assumed to be a module form the standard library, and no dependencies are generated. For programs that span multiple directories, it is recommended to pass `jcdep` the same `-I` options that are passed to the compiler.

9.2 A typical Makefile

Here is a template `Makefile` for a join-calculus program.

```
JCC=jcc
JCDEP=jcdep
INCLUDES=               #all relevant -I options here
JCCFLAGS=$(INCLUDES)    #add other options for jcc here
```

```
# prog  is composed of three units: mod1, mod2 and mod3.

# The list of object files for prog
PROG_OBJS=mod1.jo mod2.jo mod3.jo

prog: $(PROG1_OBJS)
	$(JCC) -o prog $(JCFLAGS) $(PROG_OBJS)


# Common rules
.SUFFIXES: .j .ji .jo .jio

.j.jo:
	$(JCC) $(JCFLAGS) -c $<

.ji.jio:
	$(JCC) $(JCFLAGS) -c $<

# Clean up
clean:
	rm -f prog
	rm -f *.jo *.jio

# Dependencies
depend:
	 $(JCDEP) $(INCLUDES) *.ji *.j > .depend

include .depend
```

Chapter 10

The runtime systems (jcrun)

This chapter describes the usage of join-calculus runtimes. A runtime is a command that executes bytecode files produced by the linking phase of the `jcc` compiler. By default, the standard runtime `jcrun` is used. In case additional external primitives and values are used in the bytecode file, other runtimes are called for.

10.1 Overview

The runtime support for the join-calculus comes as a family of Objective Caml executables (called *runtimes* in this chapter). This variety comes from three reasons:

- specific runtimes are needed for each machine architecture.
- runtimes comes in two Objective Caml flavors: optimized native-code (faster at run-time) or bytecode (faster at compile-time; support for graphics).
- external primitives and values used in join-calculus programs must be statically compiled and linked to the runtime. As a result, a custom runtime is generated and used for each set of externals.

All of these runtimes can work in concert as part of a distributed computation, provided that the running join-calculus programs (including mobile code) never use missing externals.

The "`jcrun`" command is the runtime that is used by default; it provides all the externals of the standard library (except module `graphics` in native code).

10.2 Usage

All runtimes have the same usage and options:

`jcrun` *bytecode-executable options*

The main argument is taken to be the name of the file containing the executable bytecode.

As described in chapter 7, the bytecode executable files produced by the `jcc` command are usually self-executable, and manage to launch a compatible runtime on themselves automatically. That is, assuming "plain.out" and "fancy.out" are bytecode executables built from the commands

```
jcc plain.j -o plain.out
jcc fancy.j -jcrun myruntime -o fancy.out
```

the commands

`plain.out` *options*
`fancy.out` *options*

are respectively equivalent to

`jcrun plain.out` *options*
`myruntime fancy.out` *options*

Notice that in case of distributed computation, it may be necessary to execute a bytecode file using another, richer runtime, such that it may host incoming mobile code that require more external primitives.

10.3 Runtime options

The runtime can run locally or participate to a distributed computation. In that case, it can run as a nameserver client or as a nameserver server.

`-l` (default)
When set, the runtime executes its bytecode locally and does not attempt to communicate with other runtimes. The nameserver primitives are not available. The runtime terminates when its runqueue become empty.

`-c` When set, the runtime executes its bytecode, and also initializes messaging in client-mode. The nameserver primitives are available. The first time they are used, a connection is attempted to another runtime in server-mode. Unless halted, the runtime never terminates.

`-s` When set, the runtime executes its bytecode, and also initializes messaging in server-mode. The nameserver primitives are locally available, and can be accessed from other runtime in client-mode that request a connection. Unless halted, the runtime never terminates.

The IP address and port number of the runtime in server-mode (used for the initial connection) may be specified using the following options:

`-port <number>`

`-host <address>`

By default, these values are respectively taken from the environment variable `JCHOST` and `JCPORT` when defined, or as the constants `localhost` and `12360`.
The other options are for internal use only.

10.4 Runtime errors

This section describes and explains the error messages that can occur at run-time. A typical error message is of the form

runtime program: *description* `(from` *originator*`)`

where *runtime* is the name of the running bytecode interpreter, *program* is the name of the join-calculus bytecode supplied as argument, *description* is the error report itself, and *originator* is the portion of the runtime code which raised the error (which is mostly used for debugging it).

10.4.1 fatal errors

Once a particular runtime reports an error, the distributed computation as a whole has failed. As it stands, only this runtime actually terminates, while other runtimes that are involved in the computation may continue. The subsequent behavior of these remaining runtimes is not specified.

command-line and options

runtime: `no bytecode executable filename`
: When running a bytecode interpreter, an executable bytecode file must be provided for local execution.

runtime: *filename*: `No such file or directory`
: When running a bytecode interpreter, the executable bytecode file *filename* given as argument does not exist.

`exec:` *filename*: `not found`
: When running a self-executable bytecode file, its default runtime interpreter *filename* could not be found in the current path. In case the program does not use external libraries, *filename* is `jcrun` and this error indicates that the Join-Calculus has not been property installed on this machine.

runtime: *filename* `is not a bytecode executable file`

: The file that the interpreter is trying to execute is not a valid executable bytecode file. Probably it has been truncated or mangled since created. Erase and rebuild it.

runtime: `cannot execute` *filename*: `external mismatch`
: The selected interpreter has not been linked with the external libraries written in Objective Caml that are called from the executable bytecode *filename*. This usually indicates the bad choice of an interpreter on the command line. When running a self-executable bytecode, this is probably because the default interpreter is not up-to-date. Clear and recompile.

runtime: `too many command-line options`
: The options on the command line are not compatible; this may be due to inconsistent mode flags `-l -s -c`, several bytecode file names, or several values for `-host` and `-port`)

local errors

runtime: `break`
The runtime was terminated by an interruption signal.

runtime: `circular migration`
A location attempted to migrate inside of one of own its sub-location. Your program should not allow such things.

runtime: `failwith` *message*
The runtime was terminated by a join-calculus message `failwith("`*message*`")`. This may be a normal termination, for instance as a result of using the `shutdown` module.

distributed support

runtime: `could not start server`
The runtime could not set-up its connection socket. Check its address.

runtime: `cannot import mobile code segment ‘‘`*segment-name*`’’: external mismatch`
The running interpreter has not been statically linked with the external libraries that are required in the code of an incoming mobile agent. Values defined in Objective Caml are not mobile; rather, their local copy is used when available. When the interpreter does not provide them, this is detected at the first attempt to migrate an offending migration. Maybe the program was compiled without the correct `-jcrun` option, or can cause unexpected migrations.

runtime: `cannot export this kind of external value`
Due to a message or a migration, the interpreter must send an external value to another runtime, but this value is not portable (cf.).

runtime: `unknown host:` *address*

runtime: `unknown service:` *port*
The runtime tried to establish a connection, either to its name server or to another runtime (e.g. to send messages on names that were received from some third party). In the first case, this is probably due to bad parameters *address* and *port* on the command line. Otherwise, there is a problem with the network (connectivity, DNS,...).

runtime: `bind: address already in use`
The runtime tried to set-up the name-server on some port that is used (or was recently used) by another process. Check for other runtimes on the machine, or use another port number.

10.4.2 warnings

The following problems are reported on standard error , but do not cause the runtime to terminate. This may change.

`Uncaught exception:` *exception-constructor* *runtime*: `exception raised during external call ''`*ocaml-expression*`''`

The Join-Calculus program (or its libraries) has tried to evaluate an external expression *ocaml-expression* defined in Objective Caml. During this evaluation, the exception named *exception-constructor* has been raised at some point and was never caught. This error is not fatal; it only causes this warning message to be locally issued, and the faulty evaluation never to complete.

Beware, some bad exception such as `Division_by_zero` may not be intercepted by the native interpreters and cause a fatal error.

Cf. documentation of Objective Caml for an explanation of particular exceptions and a discussion of bad exceptions in native-code.

NB: as long as there are no exceptions in Join-Calculus, it may be better to always consider external exceptions as a fatal errors.

`Could not establish connection to` *address*:*port* `(still trying):` *error-description*

The runtime tried to establish a connection but its system calls returned an error; this may be a transient problem (e.g. the server is not running yet) or a more serious one. In doubt, the computation goes on, and other connection attempts will occur later on.

`warning: name ''`*string*`'' not registered yet`

The name server received a `ns.lookup` request with key *string*, but found no entry in its table. This is usually due to a race-condition with another process that registers this entry; in that case, the computation will resume as soon as *string* gets registered. This may also be due to an inconsistent use of the keys, leading to a deadlock.

`warning: name ''`*string*`'' already registered (left unchanged)`

The name server received a `ns.register` request with key *string*, and there is already a value in the table. This deadlocks the call to `ns.register` and let the name table unchanged; `ns.lookup` still returns the first value.

10.4.3 bugs

All other errors are bugs, either in the runtime system or in the documentation. Please report them.