The JOCAML system documentation

Fabrice Le Fessant
Fabrice.Le_Fessant@inria.fr
INRIA Rocquencourt

May 12, 1998

Introduction

The Jocaml system is an attempt to provide all join-calculus constructs for concurrency, communication, synchronization and process mobility directly as a syntactical extension to the Objective Caml language. Consequently, this manual does not describe the Objective CAML part of the system, nor the Join-Calculus language, which are both available with separate manuals, but only how to use the Jocaml system with some knowledge of both systems.

Basically, the Join-Calculus is a special library with which Objective-Caml programs must be linked to use Join constructs. Even if the native code compiler is incorporated in the distribution, the "join" library is only available as a bytecode one. However, the `jocaml` system enables programs to run both native code and bytecode in a same runtime, with some limitations.

Finally, any question or bug report for any part of the distribution should be sent to **jc-team@inria.fr**, and not to any Objective-Caml mailing list.

Jocaml Tools

Most `jocaml` tools are Objective-Caml tools with an initial "j". For example, `ocamlc` is called `jocamlc`, `ocaml` is called `jocaml`, `ocamlrun` is called `jocamlrun`, and so on.

Suffixes are exactly the same as Objective-Caml ones. Sources files have ".ml" suffix, interfaces ".mli". Compiled files have ".cmi", ".cmo" and ".cma" suffixes. However, there is no compatibility between Objective-Caml compiled files and Jocaml compiled files, even with a same source file.

In this section, we first describe new options for old Objective-Caml tools, and then new tools designed for Join programmers.

`jocamlc`

New options:

- -join
  Link the executable with a "thread-safe" version of the standard library, needed for `join` programs.
- -nodyncheck
  Modules containing location definitions are dynamically linked during execution. Modules and primitives dependencies are resolved during the first execution of each dependencie. However, by default, these dependencies are also checked by the compiler before linking. This option prevent the compiler from checking these dependencies. This is useful if you create a location with a dependencie to a module or primitive which is not immediatly linked, but which will be available where the location will migrate.
- -make_vm
  This option enables you to create new runtime (as `jocamlrun`) to execute bytecode programs using non standard C and bytecode libraries, without using the -custom option. Executables for these new runtimes will only contain their own bytecode, while finding libraries bytecode and external primitives in the new runtime. The name of the runtime is specified with

the `-o` *runtime* option. The compiler creates two files, *runtime* and *runtime*`.cmc`. *runtime* is the runtime executable and *runtime*`.cmc` is used by the compiler to create programs executing on *runtime*.

- -use_vm *absolute-path/runtime*
  This option is used to create programs executing on *runtime*. The associated file *runtime*`.cmc` must be available in the compiler search path (the standard directory, the current one and those specified with the `-I` option). The created program will always search its runtime at the absolute position fixed by the path specified during the compilation. However, the runtime may not be at this position during the compilation.

- -cca *cclib-option*
  Creates a bytecode library (same as the `-a` option), and specifies an argument which will be passed to the C-linker when linking a program with this library. For example, the graphics library is created with:
  `jocamlc -cca "-lX11 -lgraphics" -o graphics.cma graphics.cmo`

- -noautolink
  This option prevents the compiler from using arguments given with the `-cca` option. It is most useful when, for some reasons, these arguments must be temporary overriden.

- -l *library*
  Equivalent to "*library*`.cma`"

`jocamlopt`

Mixing native code and bytecode

As an experimental feature, the Jocaml system enables you to mix native code and bytecode in a same executable. This feature is useful since join constructs are only available using bytecode, whereas good performances are achieved using native code (As an example, see `mandel.opt` in the distribution).

In a mixed runtime, native code program is started first. At the end of the native code, the bytecode program is executed. The bytecode may call native functions through dedicated "callback" modules. These modules are Objective-Caml source files with a ".mlx" suffix, which, when compiled with `jocamlopt`, generate two object files, one in native code, with real functions, and one in bytecode, which functions are only wrappers to the native code functions.

New options:

- -join
  Same as for `jocamlc`.

- -make_vm
  Same as for `jocamlc`. However, the runtime may also contain native code.
- -byte *source*
  `ml` Compile file *source*.`ml` in bytecode instead of native code.
- -cca *cclib-option*
  Same as for `jocamlc`.
- -l *library*
  Equivalent to "*library*.`cmxa`"
- -lbyte *library*
  Equivalent to "*library*.`cma`"

jocrun, joc and jocl

The jocaml system includes two runtimes: `jocamlrun` and `jocrun`. `jocrun` is created using the `-make_vm` option, and contains the thread-safe standard library, the unix library, the bytecode thread library and the join library. Thus, it can be used to execute most standard Join programs. To compile programs using this runtime, you can use the command `jocamlc -use_vm` *jocaml-bin-directory*/`jocrun` *other-args*, or use the dedicated program `joc`: `joc` *other-args*.

If you need to link your program with other libraries which are not contained in the `jocrun` runtime, you can use the command `jocamlc -join -l unix -l join` *other-args*, or the dedicated program `jocl`: `jocl` *other-args*.

Both `joc` and `jocl` programs add a `-linkall` option to include all modules in the program created. This is safer, since join programs may receive locations from other programs which have more dependencies than the original program.

joctop

The jocaml system also includes two toplevels: `jocaml` and `joctop`. `jocaml` is the Objective-Caml toplevel, without join constructs, whereas `joctop` is the Objective-Caml toplevel compiled on the `jocrun` runtime, and thus extended with join constructs (join definitions and locations).

jocns

`jocns` is a name server used by the "Ns" module. It can be started directly, or is automatically started by the first program calling Ns.register on the name server machine. The default port is 20001, but can be modified with the JNSPORT envirronment variable. The JNSNAME envirronment variable can be used to specify the name server machine if it is not the localhost.

Syntax and Examples

Extensions to Objective-Caml syntax

<table>
<tr><td colspan="3">Expressions</td></tr>
<tr><td>declaration</td><td>::=</td><td>Ocaml-declaration</td></tr>
<tr><td></td><td></td><td><code>let def</code> automatas-definition</td></tr>
<tr><td></td><td></td><td><code>let loc</code> join-locations</td></tr>
<tr><td>automatas-definition</td><td>::=</td><td>automata [<code>and</code> automatas-definition]</td></tr>
<tr><td>automata</td><td>::=</td><td>join-pattern = process [<code>or</code> automata]</td></tr>
<tr><td>join-pattern</td><td>::=</td><td>channel-decl [| join-pattern]</td></tr>
<tr><td>channel-decl</td><td>::=</td><td>synchronous-name arguments</td></tr>
<tr><td></td><td></td><td>asynchronous-name arguments</td></tr>
<tr><td>synchronous-name</td><td>::=</td><td>Ocaml-lower-ident !</td></tr>
<tr><td>asynchronous-name</td><td>::=</td><td>Ocaml-lower-lident</td></tr>
<tr><td>arguments</td><td>::=</td><td>Ocaml-pattern</td></tr>
<tr><td>process</td><td>::=</td><td></td></tr>
<tr><td></td><td></td><td>final-process</td></tr>
<tr><td></td><td></td><td>final-process | process</td></tr>
<tr><td></td><td></td><td>declaration <code>in</code> process</td></tr>
<tr><td></td><td></td><td><code>if</code> expression <code>then</code> final-process</td></tr>
<tr><td></td><td></td><td>expression ; process</td></tr>
<tr><td>final-process</td><td>::=</td><td><code>reply</code> [expression] [<code>to</code> Ocaml-lower-lident]</td></tr>
<tr><td></td><td></td><td>asynchronous-send</td></tr>
<tr><td></td><td></td><td><code>if</code> expression <code>then</code> final-process <code>else</code> final-process</td></tr>
<tr><td></td><td></td><td>{ process }</td></tr>
<tr><td>expression</td><td>::=</td><td>Ocaml-expression</td></tr>
<tr><td></td><td></td><td><code>spawn</code> { process }</td></tr>
<tr><td>asynchronous-send</td><td>::=</td><td>expression expression</td></tr>
<tr><td>join-locations</td><td>::=</td><td>location-definition [<code>and</code> join-locations]</td></tr>
<tr><td>location-definition</td><td>::=</td><td>location-name [<code>with</code> automatas-definition]</td></tr>
<tr><td></td><td></td><td>   <code>struct</code> location-items <code>end</code></td></tr>
<tr><td>location-items</td><td>::=</td><td>location-item [;; location-items]</td></tr>
<tr><td>location-item</td><td>::=</td><td>declaration</td></tr>
<tr><td></td><td></td><td>expression</td></tr>
<tr><td colspan="3">Types:</td></tr>
<tr><td>channel-type</td><td>::=</td><td>synchronous-channel</td></tr>
<tr><td></td><td></td><td>asynchronous-channel</td></tr>
<tr><td>synchronous-channel</td><td>::=</td><td>core-type => core-type</td></tr>
<tr><td>asynchronous-channel</td><td>::=</td><td><< core-type >></td></tr>
</table>

Join definitions

Join channels are defined with the `let def` construct. For example, the definition of a memory cell folows (this lines have been written in the `joctop` program):

```
# let def new_cell! s =
    let def get! () | state s = reply s |  state s
         or set! s | state _ = state s | reply ()
    in
    state s | reply (get,set)
;;
val new_cell : 'a => (unit => 'a) * ('a => unit) = chan
```

In this definition, we create a synchronous (!) name `new_cell`, with one parameter. When this name is called, it creates three names, two synchronous (!)(`get` and `set`) and one asynchronous (`state`). Notice that all names are lowercase standard Objective-Caml identifiers. Multiple join patterns in the same definition are introduced with the `or` keyword. Continuations for `reply` constructs are not specified, since there is only one synchronous name per join pattern in this example. With join patterns containing several synchronous names, continuations should have been specified with the `reply` *value* `to` *name* construct.

The type of `new_cell` is a polymorphic synchronous channel, returning a tuple of two synchronous channels. Asynchronous calls use the functionnal application construct (`state s`), but only inside a process construct. Indeed, `state s;` would raise a type error, since `state` is an asynchronous channel, used in a functionnal application inside an expression (the context before a semi colon (;) is always an expression, and not a process). In the same way, `reply` constructs and parallel compositions (|) can only appear in processes. However, the `spawn` { *process* } construct can be used to fork a process from within an expression.

```
# let (get_a,set_a) = new_cell "hello";;
val get_a : unit -> string = <fun>
val set_a : string -> unit = <fun>
```

From the syntactical point of view, there is no difference between the synchronous call (here `new_cell "hello"`) and a functionnal application. The synchronous channel type `'a => 'b` is a subtype of the functionnal type `'a -> 'b`. We also notice that that channels `get_a` and `sett_a` have now functionnal types. In fact, only names which have been defined within a join definition have the synchronous channel type:

```
# let new_cell = new_cell;;
val new_cell : 'd -> (unit => 'd) * ('d => unit) = <fun>
```

Here, the new `new_cell` name has the functionnal type, although it is equal to the old channel `new_cell`.

Finally:

```
# let _ =
    print_string (get_a());
    print_char ' ';
    set_a "world";
    print_string (get_a());
    print_newline ()
  ;;
hello world
- : unit = ()
```

Other more complicated examples can be found in the `examples/` directory of the distribution.

Location definitions and migration

Here is the definition of a location `mobile`, with two included join definitions, one with one synchronous name `go`, and another close to the previous memory cell:

```
# let loc mobile with
       go! location = Join.go location; set_position location; reply
and    set_position! location | position _ = position location | reply
    or get_position! () | position location = position location | reply location
 struct spawn { position Join.here } end
;;
val mobile : Join.location = <abstr>
val go : Join.location => unit = chan
val position : <<Join.location>> = chan
val set_position : Join.location => unit = chan
val get_position : unit => Join.location = chan
```

First, we can notice that the type of locations is `Join.location`, ie `location` from the `Join` module. The `Join` module regroups basic primitives for the join-calculus language. Here, we also use the `Join.go` function, which is used for location migration, and `Join.here`, which is the predefined toplevel location of the program.

Second, the join-calculus `init` *process* `end` construct is replaced by the `struct` *location-items* `end` construct. This construct enables you to define local values inside the location, which are not available from outside the location.

During location migration, join automatas (definitions) and Objective-Caml values behave differently. Join automatas are always uniq. Consequently, when they migrate with their location to another runtime, local pointers to these automatas from the old runtime are replaced by remote pointers to the new runtime. On the contrary, Objective-Caml values are always copied. For Objective-caml primitives and functions, this means that next call to one of them will yield to an attempt to find the function code or the primitive C code locally in the new runtime. This will success if the module where the function was defined is contained in the new runtime, or if the module code has been copied during the migration (indeed, the smallest unit of bytecode is the module, thus, all the module code migrate when the location code defined in this module migrate). This will abort and an `Reloc.ModuleNotAvailable` exception will be raised if the function code is not present. However, since the default runtime contains the standard library, the `Unix` module, the `Thread` module and the `Join` module, most needed functions are always available, and other locations dependencies should be defined in the same module as the locations.

Name server

The join library contains a `Ns` module used to make requests to the `jocns` name server. This module contains two functions:

```
val lookup : string -> 'a metatype -> 'a
val register : string -> 'b -> 'b metatype -> unit
```

`Ns.lookup` is used to lookup the value associated with a particular name in the name server, whereas `Ns.register` is used to associate a value to a name in the name server.

If no name server is running during these requests, `Ns.lookup` raises an exception, while `Ns.register` will try to start a new name server. The name server address is *hostname*:20001 by default, and can be overriden with `JNSNAME` and `JNSPORT` environment variables.

`Ns.lookup` and `Ns.register` require an extra argument of type `'a metatype` which is used to check the types of the requested value with the real type of the value in the name server. This extra argument is set with the `vartype` keyword, and a type constraint on the `'a metatype` parameter type.

For example,

```
# Ns.register "mobile" mobile (vartype:Join.location metatype);;
```

will register the previously defined location `mobile` with the name "mobile" and the type `Join.location`.

Distributed Garbage collection

A distributed garbage collector is implemented to collect unreachable objects. It is based on the SSPC garbage collector. This garbage collector can only collect acyclic distributed garbage. Distributed garbage collection is triggered by major garbage collections.

To suppress distributed garbage collection, the environment variable `JNODGC` can be set before starting the runtime. To avoid distributed garbage collections only during critical periods, use the two functions `Join.dgc_stop` and `Join.dgc_restart ()`.

Failures

The `Join` module contains two functions to handle failures:

```
val halt : unit -> 'a
val fail : location -> unit
```

`Join.halt` is used to halt a location. Currently, it does not work for the toplevel location. `Join.fail` is used to detect location failures. Indeed, a call to `Join.fail` only returns when the location in argument has failed. Currently, only locations halted with `Join.halt` are detected. Moreover, this detection is immediate for locations in the same runtime, and lazily propagated by the distributed garbage collector for different runtimes. As a consequence, only active runtimes may detect location failures.

Distributed objects

The Objective-Caml objects system has been extended with distributed objects: Objective-Caml objects are also locations. Thus, they can migrate using the `Join.go` function with all automatas, locations and objects they contain. The function `Join.goo` can be used to migrate a location into an object.

Moreover, all methods calls are implemented as a new process started in the object (as location). Thus, running method threads migrate with the object.

Exceptions

Exceptions raised in automatas'processes are propagated back to all synchronous names which will not receive a reply owing to the raised exception:

```
#let def a!() | b!() | c!() = reply to a |
   raise Not_found; { reply to b | reply to c}
;;
val c : unit => unit = chan
val b : unit => unit = chan
val a : unit => unit = chan
```

In this example, the call to `a` will receive a reply, whereas the calls to `b` and `c` will raise the `Not_found` exception.

`Join` module

Most join-calculus primitives are available from the `Join` module:

```
type location
type space_id

val new_port : int -> unit
val server : unit -> unit

val here : location
val go : location -> unit
val goo : <  > -> unit

val halt : unit -> 't
val fail : location -> unit

val getRemoteService : space_id -> string -> 'n metatype -> 'n
val getLocalService : string -> 'o metatype -> 'o
val setLocalService : string -> 'p -> 'p metatype -> unit
val getSpaceService : Unix.inet_addr -> int -> string -> 'q metatype -> 'q

val dgc_stop : unit -> unit
val dgc_restart : unit -> unit
```

Interactions with the threads library

Optimizations

Other Features