Programming in JoCaml (Tool Demonstration)

Louis Mandel[1] and Luc Maranget[2]

[1] Lri, Univ Paris-Sud 11, cnrs, Orsay F-91405
Inria Futurs, Orsay F-91893
[2] Inria Paris - Rocquencourt, Le Chesnay F-78153
{Louis.Mandel,Luc.Maranget}@inria.fr

Abstract. JoCaml is a language for concurrent and distributed programming. The language is an extension of Objective Caml with concurrent features inspired by the join-calculus.
We here present the recent release of JoCaml, motivate our fundamental design choices, compare the new release with previous ones, and give a taste of JoCaml by means of a few examples.

1 Introduction

JoCaml is a language for programming concurrent and distributed systems. It is based on ML for the computational part, and on the join-calculus for the concurrent part.

The join-calculus is a name passing calculus. The purpose of such calculi is to describe concurrent and distributed systems. Programming such systems is a different, although related, issue, since a good model offers suitable abstractions that help programmers.

Our language, JoCaml, is an extension of Objective Caml (OCaml), a popular dialect of ML. By choosing to extend an existing language, and not to design one of our own, we first intend to minimize our work. We also intend to benefit from functional programming, from pre-existing code base, and from a population of programmers open to innovation.

Up to three new keywords, JoCaml is a conservative extension of OCaml: OCaml programs retain their type and behavior. But we understand compatibility in a stronger sense: JoCaml provides a concurrent extension of ML that strictly adheres to the spirit of functional programming. Channel definitions and synchronization behaviors are programmed concisely, by the high-level join-definition concept, and declaratively, by the introduction of ML pattern matching of messages in channel definitions. Moreover, channels are typed polymorphically, as functions are in ML, types being inferred. Channels are first class-values that, amongst other things, can be passed as arguments to functions, sent as messages on channels, and occur as members of modules. This, with the polymorphic typing of channels, is our way to code re-use for concurrent components.

JoCaml web site is `http://jocaml.inria.fr/`. The site offers a source release (dating June 2007), links to articles, and a 70 pages tutorial and reference

expression ::=	*ocaml-expression*	
		join-definition
		process execution
process ::=	x(*expression*)	message sending
	reply *expression* to x	reply to synchronous channel
	process & *process*	parallel composition
	expression ; *process*	sequential composition
	let ... in *process*	local value definition
	def ... in *process*	local channel definition

Fig. 1. JoCaml syntax

manual. We have programmed a few applications in the language ourselves. Amongst those, a distributed ray tracer is the most mature. The ray tracer is available on the web site and its source code amounts to about 7000 lines.

2 The new JoCaml

The new JoCaml system is a re-implementation from scratch of the previous prototype. It focuses on compatibility with OCaml. Any OCaml source code is a valid JoCaml source code and JoCaml can also call external OCaml libraries that do not need to be re-compiled.

Briefly, we proceed by altering the OCaml compiler from parsing phase to first intermediate code generation, and by enriching the thread library of OCaml with specific support. Compiler alteration is justified by specific typing and pattern matching compilation, which both need to be perform inside the compiler. Compiler alteration is limited in the sense that we change or add a few thousand lines in the compiler original source files, add a few source files, and retain the OCaml formats for binary files.

Our focus over compatibility and limited alteration of OCaml, made us abandon the mobility features of the join-calculus. Nevertheless, there are useful distributed programs that can be written without code mobility.

Moreover, the new JoCaml extends the synchronization mechanism of the join-calculus with pattern matching. It allows to define synchronization not only on the presence of a message on a channel, but also on the value of the message.

3 A join-definition

JoCaml adds the new syntactical category of *processes* to OCaml syntax (Fig. 1). In contrast to expressions processes yield no result and execute asynchronously. Additionally, JoCaml slightly extends OCaml expressions. The `spawn` *proc* construct introduces processes in expressions: *proc* is executed asynchronously and spawn returns immediately.

The join-definition is the distinctive feature of the join-calculus: it defines several channels and their reception behavior at the same time. In JoCaml, join-definitions are introduced by def and can occur both in processes and expressions. We illustrate join-definitions by the example of a concurrent buffer based on the two-lists implementation of functional FIFO queues.

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

let create_buffer () =
  def state(xs,ys) & put(x) = state(x::xs,ys) & reply () to put
   or state(xs,y::ys) & get() = state(xs,ys) & reply y to get
   or state(_::_ as xs,[]) & get() =
     state([], List.rev xs) & reply get() to get
  in
  spawn state([],[]) ;
  {put=put; get=get;}
```

Our buffers are records, a pure OCaml concept, the novelty resides in the join-definition (def... in above). Three channels are defined: `state`, `put` and `get`. Channel `state` is *asynchronous*. Message sending on an asynchronous channel is an elementary process, as illustrated by `spawn state([],[])` above, for instance. By contrast, `put` and `get` are synchronous channels. Message sending on a synchronous channel yields a result, and thus is an expression. In fact, to the sender, synchronous channels behave as functions and have functional types.

The behavior of the buffer is expressed by three *reaction rules* that compete (or) for consuming messages. A reaction rule consists in a *join-pattern* and in a *guarded process* (separated by =). The semantics is as follows: when there are messages pending on all the channels in the join-pattern and they match the patterns present as formal arguments, then the guarded process may be fired. The guarded process is executed asynchronously, but may transmit return values to the callers of synchronous channels (reply/to).

The idea of the buffer is to store the FIFO queue (implemented by a pair of lists) as a message on the channel `state`. By the organization of join-patterns, which all include `state`, and the fact that there is at most one message on this channel, exclusive access to the internal state of the buffer is granted to the callers of synchronous `put` and `get`.

The first join-pattern `state(xs,ys) & put(x)` is satisfied whenever there are messages on both `state` and `put`. The behavior of the guarded process is to perform two actions in parallel (`&` in processes): (1) send a new message on `state` where the value `x` is added to the list `xs` and (2) return the value `()` to the caller of `put`.

The second join-pattern `state(xs,y::ys) & get()` is satisfied when there are messages on both `state` and `put` *and* that the message on `state` matches the pattern `(xs,y::ys)`. That is, the message is a pair whose second component is a non-empty list. The process guarded by this join-pattern removes one value from the buffer and returns it to the caller of `get`. The last join-pattern `state(_::_ as xs,[]) & get()` is satisfied when there is a message on `get` and

a message on `state` that matches a pair whose first component is a non-empty list and second component is an empty list. The corresponding guarded process transfers elements from one end of the queue to the other and performs `get` again. Notice that there is no join-pattern that satisfies `state([],[]) & get()`. As a consequence, a call to `get` is blocked when the buffer is empty.

To initialize the buffer, a message `([],[])` is sent on `state`. The `spawn` construct is here necessary, since the message sending appears in expression context (the body of the function `create_buffer`).

4 Distributed computation

The join-calculus provides a transparent model for distributed computation. Guarded processes always execute on the site where they are defined but can be fired from any site. More precisely given a channel c, the sending of a message on c can be performed on any site (provided c is known), while the reception on c can occur only on the site where c is defined. This is by design, and comes in sharp contrast to the model of the π-calculus, where it is sufficient to know c to perform emission and reception on c.

Obviously, the join semantics is much easier to implement than the π semantics in a distributed setting. Basically, message sending to a remote site decomposes into a transport phase and a synchronization phase (join-pattern matching), the latter being performed locally on the receiving site.

However, performing the transport phase (and the related global naming of sites and channels) does not upgrade concurrent JoCaml into distributed JoCaml as if by magic. Two important issues arise that are not really expressed in the join model: channel publication and failures. We addressed those pragmatically, so as not to delay the release of the new JoCaml.

When they start, sites (JoCaml programs) have nothing in common. But, so as to initiate communication, sites need to share at least a few channel names. To that aim, JoCaml provides a *name service* that basically is a repository of channel names, indexed by plain strings. In contrast to the JoCaml language, there is no type safety at all. As to failures, our treatment is rather unsophisticated as we rely exclusively over direct routing: communicating sites are connected by a bi-directional link (a TCP socket). Then, the failure of the link, is interpreted by one partner as the failure of the other partner. We plan to improve these two points in future releases.

5 Conclusion

JoCaml is one amongst many recent language that offer serious support for concurrency and distribution (Erlang, Cω, Alice, Scala to cite a few). In our view, JoCaml main contribution resides in the programming style it favors: a smooth integration of functional programming for concurrent and distributed applications. Our tool demonstration will focus on this point.