Written Presentation: JoCaml, a Language for Concurrent Distributed and Mobile Programming

Nicolas Bettenburg

Universitaet des Saarlandes, D-66041 Saarbruecken, nicbet@studcs.uni-sb.de

Abstract. As traditional programming languages are designed for closed, sequential architectures, concurrent and distributed programming is either hard to achieve or error-prone. JoCaml as a high-level language was designed with dedicated support for concurrency, synchronization as well as the support for distributed execution of programs. JoCaml itself is based on the Join Calculus and extends the OCaml language, keeping all of it’s features. This written presentation will show how JoCaml can be used to achieve concurrent and distributed programming, with the main focus on concurrency.

1 Introduction

Distributed, as well as concurrent programs, are usually hard to write and understand - even harder to debug or proved to be correct due to asynchrony and non-determinism. JoCaml is an attempt to provide a functional high-level language with support for distributed and concurrent programming. Traditional high-level distributed programming languages rely heavily on scripting languages, which are often specialized and lack elements of structure like modules, classes or user-defined types, crucial for flexible programming of large projects.

JoCaml is based on the OCaml programming language and thus inherits all of it’s features, which make it a feasible setting for distributed programming.

– Static typing: Important for distributed programming, since debugging runtime errors is very hard to cope with.
– Byte-code compilers: Providing separate compilation and flexible linking as the key part for JoCaml’s implementation of code mobility at runtime.
– Low-level support: Good support for low-level system programming.

JoCaml extends OCaml in such way, that OCaml programs and libraries are just a special kind of JoCaml programs and libraries (OCaml ⊆ JoCaml).

2 Extending OCaml with support for Concurrent Programming

OCaml is a sequential programming language, so every expression is executed in a deterministic way in call-by-value. Since concurrency is desired, the first
extension of JoCaml to OCaml is the support for lightweight concurrency, message passing and a mechanism for message-based synchronization. JoCaml at this point introduces a new expression called `spawn` with the syntax:

```
spawn process; expression
```

executing process and evaluating expression in parallel. This means, that the operations in process and expression run independently (concurrent) in a non-deterministic way.

Since OCaml only has support for expressions, JoCaml introduces a new syntactic class `process` which is recursively defined with expressions, handing the real mapping of the processes to system threads over to the JoCaml compiler and runtime.

There is another thing, special to processes: they are not meant to return a result; their only means of interaction is sending asynchronous messages on channels. Such an asynchronous message is a process itself. In order to do so, JoCaml introduces `channels` and `local channel definitions` for processes: they are first-class values with a communication type used to form expressions and send messages. Channel definitions bind names with a static scope and attach guarded processes with these names. By passing a message over these names, a copy of the guarded processes is executed.

In order to provide synchronization facilities, JoCaml holds on to the ML paradigm: definition by pattern-matching, to provide a declarative way of specifying inter-process synchronization. This in fact does not export state: it leaves the state inside the process. This is done by allowing the joint definition of a number of channels by matching concurrent message patterns on these channels. This way of synchronization via patterns was first introduced in the join calculus \(^1\) in 1996 [1] and has several advantages for compilation efficiency and efficient implementation of routing.

3 Extending OCaml with support for Distributed Programming

We have previously seen how concurrency was introduced to OCaml. This section will show how this can be used across several machines on an asynchronous network with distributed message passing and even process mobility. The programming model proposed is based on the Join Calculus as stated earlier before and is characterized by an explicit notion of locality: since there is the need to represent a set of runtimes with their local processes on a network, the join calculus defines a basic unit of locality - called `location`.

- Locations have a `nested structure` (so they can contain sub locations); in fact, a whole JoCaml program itself is a location called the `root location`.

\(^1\) The Join calculus itself is based on the π-calculus [2] which was originally proposed by R. Milner as a progress to the original λ-calculus
A configuration of machines, distributed over the network and executing JoCaml programs can thus be seen as a location tree; each location having it’s own definitions and processes.

- Locations are transparent; channels have a global lexical scope, so any process that has received a channel name can use it independently of the location that defined the channel name.
- Locations form units of mobility; At any time, a location (together with it’s content) can be migrated from one machine to another. Location names can be passed in messages and the be used as target addresses for such migrations.
- Locations represent atomic units of failure; They can be used to detect failure, halt the execution of all the location’s content or implement failure recovery mechanisms.

4 Concurrent programming in JoCaml

As addressed in section 2, JoCaml extends the OCaml language with some new language features, which enable concurrent programming. We will now take a closer look on these extensions.

4.1 Channels in JoCaml

Channels (also called port names) are the main new primitive values in JoCaml. There are two kinds of channels: synchronous and asynchronous, depending on their usage for communications. In either flavor, a new channel is introduced by a new let def binding; the right hand side of the channel definition is the process fired for every message sent via the channel name.

Asynchronous channels are defined by

```ocaml
# let def channelname! varname = process
val channelname: <<type>>
```

Syntactically, the presence of a ! in the definition of a channel’s name indicates it’s asynchrony. The channel has type <<x>>, where x is the type of the values the channel carries. Since it is an asynchronous channel, the execution of process is concurrent.

Synchronous channels are defined by

```ocaml
# let def channelname varname = process; reply
val channelname: a -> b
```

Syntactically, the absence of a ! in the definition of a channel’s name indicates it’s synchrony. The channel has type a -> b which reminds of a function. The difference is, that a synchronous channel must explicitly send back some values as results using reply - a function implicitly returns the value of their main body.
Message sending on asynchronous channels appears in processes, message sending on synchronous channels appears in expressions (as if they were functions). This partition of channel usage is one possible explanation for the design decision to have two different types for asynchronous and synchronous channels (since the type checker should flag an error whenever a channel is used in the wrong context).

Since channels are first-class values in JoCaml, they can also be sent and received in messages (often referred to as name mobility[2]) which adds to the expressiveness of JoCaml. One can write higher-order functions and ports (for example turning a function into an asynchronous channel) or have polymorphic types for channels.

The channels introduced to JoCaml are uni-directional channels. However - using a concept called join-patterns, one can also define bi-directional channels in JoCaml.

4.2 Expressions in JoCaml

Expressions are the same as they were in OCaml: they are executed in a synchronous, deterministic call-by-value manner and produce a value when they finish. The most basic expression sends a message on a synchronous channel and waits for the result (blocks). For example, the expression

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

sends two times on a synchronous channel called print and always evaluates to the empty result (since the print channel has type int->unit in this case) while outputting 12.

4.3 Processes in JoCaml

Processes are the main new syntactic class in JoCaml. Since only declarations and expressions are allowed at top-level of a JoCaml program, JoCaml provides the spawn keyword to turn a process into an expression. The most basic process sends a message on an asynchronous channel.

spawn { ... }

Processes can be group by using braces “{ }” and compositied for concurrent execution via “|”, for example

spawn { ... { ... | ... } | ... }

Process composition also includes conditionals (if then else), functional matching (match with) and local binding (let in, let def in). Sequences may also appear inside processes, with the general form expression; process. This form is due to the fact, that processes do not return values - so the value generated by expression must be discarded. As Processes are executed concurrently, unlike expressions, the code
# spawn {echo 1 | echo 2}

produces two possible outputs: either 12 or 21 depending on which of the processes ( {echo 1} or {echo 2} ) was executed first.

## 4.4 Synchronization Patterns

Join patterns, as introduced with the join calculus\[1\] extend channel name definitions with synchronization. Such a pattern defines a set of channels at once and specifies a synchronization condition to receive messages on these channels. For example, in

```ocaml
#let def channelnameX! var1 | channelnameY! var2 = process
```

there must be messages on channelnameX and channelnameY to trigger the execution of the guarded process on the right-hand side. When multiple message matches are available, which message will be consumed is non-deterministic. Once again, the composition operator “|” is used to form the join patterns.

- Join patterns are the programming paradigm for concurrency in JoCaml. They allow the encoding of many concurrent data-structures.
- Join patterns can mix up synchronous and asynchronous channel definitions.
- If multiple synchronous channels are defined in a join pattern, each `reply` construct must specify the name to which it replies via `reply to name`.

```ocaml
# let def channelX var1 | channelY var2
  = reply to channelX | reply to channelY
```

- Several join patterns can be co-defined with the keyword `or`.

```ocaml
# let def channelX! var1 | channelY! var2 = process1
  or channelX! var1 | channelZ! var3 = process2
```

With join patterns, many common programming styles, either sequential or concurrent, can be expressed. For example explicit synchronization points in the parallel execution of tasks, a so-called synchronization barrier can be expressed with the usage of two synchronous channels in a join pattern:

```ocaml
#let def synch1 () | synch2 () = reply to synch1 | reply to synch2
val synch1: unit->unit
val synch2: unit->unit
```

This pattern can now be used for example for process interleaving, by spawning to processes that alternately send an empty message on synch1 and synch2.

The generalization of synchronization barriers is the `Join/Fork Parallelism`, which defines a variant, that performs to computations in parallel and then joins the results.
5 Join-Calculus and other programming languages

The concepts of explicit locality and join patterns are not restricted to JoCaml, or the functional world at all. They can even be transferred to an imperative setting like C# [3]: Processes are here so-called asynchronous methods:

```csharp
async myMethod(...) {
    // Method Body
}
```

The execution of the method body is scheduled in a different thread. Again, asynchronous methods (like processes) do not produce a result value, so the `void` keyword is substituted by the `async` keyword.

Join Patterns are here so-called chords:

```csharp
class myClass {
    string Get() & async Put(string s) {
        return s;
    }
}
```

In this chord, execution of method Get() blocks until a string is provided via put (if not before). Detailed information on Polyphonic C# can be found in [3].

6 Conclusion

We have seen that the Join Calculus is a nice base for a new type of programming model for concurrent and distributed programming. It is not restricted to the functional world, however, OCaml as the base language with JoCaml as it’s extension simplifies concurrent programming (in regard of traditional lock-based approaches). As a difference to another approach to concurrent programming, we have seen in the Seminar: Transactional Memory, the programmer in JoCaml has to keep concurrency in mind while writing the program, where as with the transactional memory approach, he could later on make already existing code safe for concurrency by using atomic. Nonetheless, JoCaml also has some draw-backs, which are actually based on the way locations are handled: the programmer has to define suitable portions of code (locations) - suitable in a regard to error handling. Asynchronous processes just print an exception to stdout, synchronous processes terminate with the exception instead of the reply, but the complete failure recovery mechanism is left to the programmer.

References