Seminar: Advanced Functional Programming

JoCaml: A Language for Concurrent Distributed and Mobile Programming

Nicolas Bettenburg

Universität des Saarlandes
Saarbrücken

22.03.2006

Overview

Concurrency: Definition and Concerns

Concurrency

Property of systems which consist of computations that execute overlapped in time, and which may permit the sharing of common resources between these computations.

- Multiple Concurrency Models
  - Lock-Based Approach
  - Transactional Memory (as seen in Seminar)
- Race Conditions, Deadlocks, Starvation
- Debugging, Correctness

New Approach: JoCaml

- Underlying Concurrency Model: Join Calculus (1996)
- Based on Objective Caml
  - Statically typed language
  - Byte-code compiler (code mobility)
  - Good system programming support
  - Efficient Garbage Collector
  - sequential, call-by-value evaluation, deterministic
- Extension maintains original language features.
- JoCaml extends OCaml with support
  - for lightweight-concurrency
  - Message Passing
  - Message-based Synchronization

Expressions

Expressions

- Executed synchronously.
- Every Ocaml expression is a Jocaml expression

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

Processes

Processes

- Executed asynchronously
- No result value
- Communicate by sending messages on channels.

```
# spawn { echo 1 };;      can also be written as
# spawn { echo 2 };;      # spawn {echo 1 | echo 2};;
=> 12 (or 21 !!)
```

Channels

Uni-Directional Channels

- Synchronous, in expressions, send and await answer (block).
- Asynchronous, in processes, send message.

```
# let def my_chan_sync x = print_int x; reply;;
val my_chan_sync: int -> unit

# let def my_chan_asynch! x = print_int x; ;;
val my_chan_async <<int>>
```

Synchronization and Concurrency Control

Synchronization by Pattern Matching

Join patterns extend port name definitions with synchronization.

```
# let def fruit! f | cake! c = print_string(f^" "^c); ;;
# spawn{ fruit orange | fruit apple | cake sacher};;
```

Synchronization Barriers

Represent explicit synch-points also know as rendez-vous.

```
# let def sync1 () | sync2 () = reply to sync1 | reply to sync2;;
# spawn {for i=0 to 9 do sync1(); print_int 1 done;};;
# spawn {for i=0 to 9 do sycn2(); print_int 2 done;};;
=> 12121212121212121212
```

A reference cell

```
# type 'a jref = {set: 'a -> unit; get: unit -> 'a}
# let def new_ref u =
#       let def state! v | get () = state v | reply v
#       or state! v | set w = state w | reply
#       in state u | reply{get=get; set=set}
# let r0 = new_ref 0 ;;
type 'a jref = { set: 'a->unit; get: unit->'a}
val new_ref : 'b -> 'b jref
val r0 : in jref
```

- internal state of cell = content
- lexical scoping keeps state internal
- content stored as message v on channel state

Distributed Model in JoCaml

Distributed Programming

Distributed Programming is the execution of computations on one or more machines that share their resources.

- Any machine may join or quit the computation.
- At any time, every process or expression is running on a given machine.
- They may migrate from on machine to another.
- System-Level processes communicate via TCP/IP over the network.

Nameserver

The Nameserver

Used to bootstrap a distributed computation. A built-in library that exchanges a few channel names.

- Needed since JoCaml has lexical scoping.
- Function to register a channel in a global table.
- Function to look-up a value in the global table.

```
# spawn{ let def f x = reply x*x
         in Ns.register "square" f vartype; };;
```

Mobility: Locations and Mobility

Locations

Represent units of mobility.

```
# let loc here
#       def square x = reply x*x
#       and cubic x = reply (square x)*x
# do {print_int (square 2); } ;;

# let loc mobile
# do {
#       let there = Ns.lookupo "here" vartype in go there;
#       let sqr = Ns.lookup "square" vartype in
#         let def sum (s,n) =
#            reply (if n=0 then s else sum (s+sqr n, n-1)) in
#              print_string (sum(0,5)); } ;;
```

Termination and Failure (Recovery)

- Some parts of distributed computation may fail.
- Detect failures and take adequate measures
  - Cleanly report the problem
  - Abort related parts of computation
  - Make another attempt on a different machine
- a location can run a halt() process
- a location can detect if another location has halted
- Up to the programmer to define locations as suitable units of failure recovery !
- Up to the programmer to provide a recovery mechanism !

Summary

- Based on Join Calculus
- Nice extension of OCaml
- Idea of join calculus also applicable to other languages like C Sharp.
- Different Model than Memory Transactions. (atomic vs. joins)
- Programmer has to consider concurrency while writing application.
- Distributed Programming based on concurrency.

List of References

- F. Le Fessant, C. Fournet, L. Maranget and A. Schmitt: `JoCaml: a Language for concurrent Distributed and Mobile Programming.` [AFPS 2002]
- C. Fournet and G. Gonthier: `The join calculus: a language for distributed mobile programming.` [APPSEM 2000, LNCS vol 2395 p. 268-332]

Questions

Thank you for your listening.
Questions?