Explicit concurrent programming in high-level languages

Kari Kähkönen

March 19, 2009

1

Outline

Contents

2

1 Introduction

Explicit parallelism

- In languages that use explicit parallelism the programmer must explicitely define which parts should be executed as independent parallel tasks
- The programmer has complete control over the parallel execution
- This is opposite to implicit parallelism where the system decides automatically which parts to run in parallel

3

2 Join calculus

2.1 Overview

Join calculus

- Join calculus aims to support asynchronous, distributed and mobile programming
- Join operational semantics are specified as a reflexive chemical abstract machine (CHAM)
- Using CHAM the state of a system is represented as a “chemical soup”

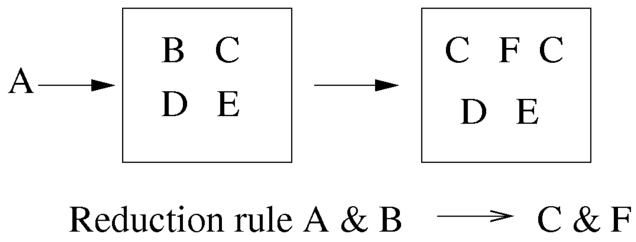
- active definitions
- running processes
- a set of reduction rules

- Join calculus can be seen as a functional language with Join patterns (provides synchronization between elements in the “soup”)

4

The basic idea behind join calculus can be understood through reflexive chemical abstract machine. The discussion here about CHAM (or RCHAM) is informal but a more formal explanation of it and join calculus can be found in [3]. In join calculus processes can create new processes and new active definitions to the chemical solution. In addition to these we have reduction rules that state on the left hand side of the rule what elements need to be in the chemical solution in order to create the elements on the right hand side of the rule. When a reduction rule is applied, the elements corresponding to the left hand side are removed and the elements on the right hand side are added to the solution. The following example picture illustrates this further.

CHAM example



5

2.2 JoCaml

JoCaml

- JoCaml = Objective Caml + Join calculus
- The programs are made of processes and expressions
- Channels (also called port names) are the main new primitive values compared to Objective Caml
- Processes can send messages on channels

6

JoCaml [2] is a functional language that has been derived from Objective Caml. It has adopted join calculus for concurrent and distributed programming. The following examples illustrate the use of join calculus in JoCaml. Most other language features are left outside this discussion. A good introduction to JoCaml can be found from [2] although it is recommended that some knowledge over Objective Caml should be obtained before going deeply into JoCaml extensions.

Channels and processes

- Channels are created with a def binding

```
#def echo(x) = print_int x; 0
```

- echo is an asynchronous channel → sending a message on it is a nonblocking operation and it cannot be said when the printing actually happens

- Processes are created with a keyword spawn
- There can be concurrency inside processes as well

```
#spawn echo(1) & echo(2)
#spawn begin
    print_int 1; print_int 2; 0
 end
```

7

In the example above the use of the def binding can be considered as defining a reduction rule for the chemical abstract machine. For example, if a term echo(1) is added to the chemical solution, the defined reduction rule allows it to be removed from the solution and a new process that prints one to be added.

Channels and processes

- The process created by sending messages are called guarded processes and they can spawn new messages

```
#def echo_twice(x) = echo(x) & echo(x)
```

- Channels can take tuples as arguments and even other channels as well

```
#def foo(x,y) = echo(x) & echo(x+y)

#def twice(f,x) = f(x) & f(x)
#spawn twice(echo, 5)
```

8

Channels are considered first-class values and therefore they can be sent as messages.

Synchronous channels

- Synchronous channels can be used to define processes that return values
- Synchronous channels use reply/to constructs

```
 #def fib(n) =
    if n <= 1 then reply 1 to fib
    else reply fib(n-1) + fib(n-2) to fib

#print_int (fib 10)
>89
```

- In the example above the synchronous channel behaves like a function but the real value of them comes apparent when used with join patterns

9

Join patterns

- Join patterns define multiple channels and specifies a synchronization pattern between them

```
#def foo() & bar(x) = do_something(x) ; 0
```

- In the example above messages to both foo and bar must be sent before the guarded process is executed

```
#def a() & c() = print_string "ac" ; 0
  or b() & c() = print_string "bc" ; 0

#spawn a() & b() & c()
```

- The example above illustrates a composite join definition
- Channel c is defined only once and can take part in either synchronizations

10

Before this point we had only considered cases where the reduction rules in a chemical solution contained only one term on the left hand side. This in itself is not very interesting but with join patterns we can define more generalized reduction rules. This allows us to write interesting concurrent programs as will be illustrated by the examples on the following slides.

The in the composite join definition example above, the result of sending messages to channels a, b and c concurrently will result in either printing ac or bc. If the example was changed so that a second message to channel c was sent as well, the program would print both messages. In this case it would not be determined in which order the print expressions are executed.

Mutual exclusion example

- Using both asynchronous and synchronous channels allows us to define many concurrent data structures such as the counter bellow

```
#def count(n) & inc() = count(n+1) & reply to inc
  or count(n) & get() = count(n) & reply n to get
#spawn count(0)
```

- A safer way to define a counter would be:

```
#let create_counter () =
  def count(n) & inc0() = count(n+1) & reply to inc0
   or count(n) & get0() = count(n) & reply n to get0 in
  spawn count(0) ;
  inc0, get0
#let inc,get = create_counter()
```

11

The counters defined above can be used consistenly by several processes. This is because when a process gets or increments the counter, it will "take" the counter for itself and put it back only after the operation has finished. The first variant is somewhat dangerous in the sense that if more than one invocations on count are made, the mutual exclusion will no longer work. In the second variant this is avoided by hiding the count behind the lexical scope of create_counter.

Control structures

- Many common synchronization primitives can be expressed with Join patterns
- Locks:

```
#let new_lock () =
  def free() & lock() = reply to lock
  and unlock() = free() & reply to unlock in
  spawn free() ;
  lock, unlock
#let my_lock,my_unlock = new_lock()
```

12

Control structures

- Barriers:

```
#def join1 () & join2 () = reply to join1 & reply to join2

#spawn begin
    (print_int 1 ; join1 (); print_string "a" ; 0)
  & (join2() ; print_string "b" ; 0)
 end
```

- Asynchronous loops:

```
#def loop(a,x) = if x > 0 then (a() & loop(a,x-1))
```

13

The barrier example can print either 1ab or 1ba depending on the scheduling. In the asynchronous loop example, we can concurrenly execute a guarded processes corresponding to a() if the order of these iterations is irrelevant to us.

Timeouts

- The following example illustrates how we do not have to wait for a result of some computation if it takes too long

```
#let timeout t f x =
  def wait() & finished(r) = reply Some r to wait
  or  wait() & timeout() = reply None to wait in
  spawn begin
    finished(f x) &
    begin Thread.delay t; timeout() end
  end ;
  wait()
```

- In this example the computation of f does not stop after the timeout. Exceptions could be used to archieve this.

14

2.3 Other languages

Other languages

- Join calculus has been incorporated into other languages as well, e.g., Join Java and Polyphonic C#

- Join Java adds Join patterns and a new signal return type to Java

```
final class SimpleJoinPattern {
    int A() & B() & C(int x) {
        return x;
    }
}
final class SimpleJoinThread {
    signal athread(int x) {
        ...
    }
}
```

15

The method in the first example is executed only after calls are made to all the three methods in the join pattern. The call to A will block if the other two methods are not called as it will return a value (i.e., it is a synchronous channel). The second example shows how an asynchronous method can be written. With the signal keyword, the called method will be executed in a new thread and the calling thread can proceed as normal (i.e., it will not block).

More information about Join Java can be found in [4].

3 Actor model

3.1 Overview

Actor model

- In Actor model all the computation is done by actors.
- Actors can concurrently
  - send messages to other actors
  - create new actors
  - designate the behavior that is used when the next message is received
- All communication is done asynchronously
- Actors are identified by addresses and messages can only be sent to known addresses

16

The Actor model was introduced by Carl Hewitt, Peter Bishop and Richard Steiger in [5] and in it the concurrent computation is done by using actors as the universal primitives. It can be seen that in the Actor model everything is an actor in a similiar sense than in some object-oriented languages everything is an object.

As all communication is asynchronous, sending messages never causes an actor to block. However, the actors can stop to wait for a new message to be received.

Actor model

- There is no requirement that the messages arrive in the order they are sent
- In this sense sending messages is similar to sending IP packets
- As different processes communicate only using message passing, there is no need for locks
- Actor model (or some of its variations) is employed in multiple programming languages
  - Erlang
  - Act 1, 2 and 3
  - ActorScrip
  - etc.

17

3.2 Erlang

Erlang

- Erlang is a general purpose functional programming language that uses Actor model for concurrency
- It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications
- Erlang processes are lightweight processes (not operating system processes or threads) that have no shared state between them
- Supports hot code loading

18

The processes in Erlang can be considered as the actors in the Actor model and they are essencially user-space threads (somewhat similar to green threads [6]). The code in Erlang is managed in modules and the language supports hot code loading by allowing two versions (new and old) of a module to be kept in the memory. The old module must make an explicit call in order to move to using the new version.

Processes

- A process is a complete virtual machine
- A process can create another one using keyword spawn

```
Pid2 = spawn(Mod, Func, Args)
```

- Pid2 is the identifier of the new process and it is known only to the creating process
- self() can be used to return the identifier of the executing process

19

Message passing

- In the example bellow, Msg is a variable and is bound when a message is received
- Variables can be bound only once
- Note that Pid2 in receive part has already been bound

```
-module(echo).
-export([go/0], loop/0).
go() ->
        Pid2 = spawn(echo, loop, []),
        Pid2 ! {self(), hello},
        receive
                {Pid2, Msg} ->
                  io::format("P1 ~w~n", [Msg])
        end,
        Pid2 ! stop.
```

20

Message passing

```
[example continued from the previous slide]

loop() ->
        receive
                {From, Msg} ->
                  From ! {self(), Msg},
                  loop();
                stop ->
                  true
        end.
```

21

More on message passing

- Lets assume that two processes send messages a and b to a third process (a and b are atoms, Msg is a variable)
- To receive a before b (regardless of the send order):

```
receive
    a -> do_something(a);
end,
receive
    b -> do_something(b);
end
```

- To process the first message to arrive:

```
receive
    Msg -> do_something(Msg);
end
```

22

Registered processes

- Keyword register can be used to register a process identifier with an alias
- Any process can send messages to a registered process

```
start() ->
        Pid = spawn(num_anal, server, [])
        register(analyser, Pid).
analyse(Seq) ->
        analyser ! (self(), {analyse,Seq}},
        receive
            {analysis_result, R} ->
                R
        end
```

23

The example above also illustrates the client/server model. Basically a server is started and then clients can send messages to it, do something while the server processes the message, and when the result is needed, the client can wait for the result to be sent back (or just read it if it has already been sent).

Timeouts

- The example bellow performs do_something if a message is received before T ms has elapsed

```
time_example(T) ->
         receive Msg -> do_something(Msg);
         after T -> do_something_else();
end.
```

- The message buffer can be flushed followingly

```
flush ->
      receive Any -> flush();
      after 0 -> true
end.
```

24

4 Data-flow programming

4.1 Overview

Data-flow programming

- Data-flow programming provides automatic synchronization by introducing (concurrent) logic variables and futures (the names may vary from one language to another)
- Logic variables are initially unbound
- Accessing an unbounded logic variable automatically suspends the executing thread
- It is not possible to change the value of a logic variable after it has been bound
- A future is a read only capability of a logic variable
- Data-flow programming allows programmers to focus on what needs to be synchronized

25

The data-flow approach presented here has been influenced by concurrent logic programming [7] and concurrent constraint programming [8]. The basic idea behind it can be seen when considering an example. Let us assume that a main thread creates a logic variable and passes it to a newly created thread. The new thread does some computations and finally stores the result to the logic variable. The main thread in the meanwhile can continue its own computation until it needs the result from the new thread. When the main thread tries to read the value from the logic variable, it will automatically block if the new thread has not yet bound a value to it. Otherwise the main thread can read the value normally. In this simple case the programmer had only to focus on the one variable that had to be synchronized.

4.2 Flow-Java

Flow-Java

- Flow-Java is a conservative extension of Java
- Adds single assignment variables (variant of logic variables) and futures
- Overhead for the runtime is in most cases between 10% and 40%
- Single assignments are introduced with the type modifier single
- A single assignment variable can be bound by using `@=`
- Aliasing is possible and equality testing has also been extended

```
single Object s;
Object o = new Object();
s @= o;
```

26

A good overview of Flow-Java and its implementation can be found from [9].

Aliasing single assignment variables is possible while they are still unbound. Binding a single assignment variable also binds their aliases. Equality tests return true if two unbound single assignment variables are aliased. Otherwise the test suspends until the variables become aliased or bound.

Example

```
class Spawn implements Runnable {
   private single Object result;
   private Spawn(single Object r) {
      result = r;
   }
   public void run() {
      result @= computation();
   }
}
public static void main (String[] args) {
   single Object r;
   new Thread(new Spawn(r)).start();
   System.out.println(r);
}
```

27

Futures

- In the previous example, the main thread can unintentionally bind the result
- To prevent this, futures can be used
- The future of a single assignment variable is obtained by a conversion from single t to t
- Implicit conversion allows integration with normal Java

```
public static Object spawn() {
   single Object r;
   new Thread(new Spawn(r)).start();
   return r;
}
```

28

Barrier example

```
class Barrier implements Runnable {
   private single Object left;
   private single Object right;
   private Barrier(single Object l, single Object r) {
      left = l; right = r;
   }
   public void run() {
      computation();
      left @= right;
   }
```

[continues on the next slide...]

29

Barrier example

```
public static void spawn(int n) {
   single Object first; single Object prev = first;
   for(int i = 0; i < n; i++) {
      single Object t;
      new Thread(new Barrier(prev, t)).start();
      prev = t;
   }
   first == prev;
}
```

30

4.3 Oz

Oz

- All variables in Oz are logic variables (also called dataflow variables)
- Executing a statement in Oz proceeds only when all real dataflow dependencies on the variables involved are resolved
- Oz is a concurrency-oriented language
- Threads are cheap to create in Mozart (60 times faster than in Java 1.2)
- All threads are run by Oz emulator (the main system thread of the process)
- Mozart Programming System is an implementation of Oz

31

The primary implementation of Oz is Mozart and it can be found from [1] together with good documentation.

A simple example

- **thread ... end** forks a new thread

```
declare X0 X1 X2 X3 in
thread
   local Y0 Y1 Y2 Y3 in
      Y0 = X0+1
      Y1 = X1+Y0
      Y2 = X2+Y1
      Y3 = X3+Y2
      {Browse [Y0 Y1 Y2 Y3]}
   end
end
```

32

In this example the execution blocks before the first assignment statement as X0 does not have a value. When this variable is given a value, the execution can proceed to the next assignment (which also blocks if X1 does not have a value). The browse command simply opens a graphical window and prints the values of the Y-variables.

A concurrent map function

- The following function generates a new list by mapping function F to its each element
- Each element is processes in a new thread

```
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then thread {F X} end | {Map Xr F}
   end
end
```

33

Streams

- Threads can communicate through streams in a producer-consumer way

```
fun {Generator N}
   if N > 0 then N|{Generator N-1}
   else nil end
end
local
   fun {Sum1 L A}
      case L
      of nil then A
      [] X|Xs then {Sum1 Xs A+X}
      end
   end
in fun {Sum L} {Sum1 L 0} end
end
{Browse thread {Sum thread {Generator 100} end} end}
```

34

It is important to notice in this example, that the thread using the list generated by the generator (or producer) can start using the list even before the generator has finished.

Synchronizing the streams

- In the previous example the communication was asynchronous
- If the producer works faster than the consumer, more and more memory is needed for the buffering
- One way to solve this is to use futures and ByNeed primitive
- ByNeed takes a one-argument procedure as argument and returns a future
- If this future is accessed, the procedure given for ByNeed is used to bind a value to the future

35

Example with futures

```
local
   proc {Producer Xs}
      Xr in
      Xs = volvo|{ByNeed {Producer Xr} $}
   end
   proc {Consumer N Xs}
      if N>0 then
```

```
         case Xs of X|Xr then
            if X==volvo then
               {Consumer N-1 Xr}
            else {Consumer N Xr} end
         end
      end
   end
in
   {Consumer 1000000 thread {Producer $} end}
end
```

36

Questions?

37

References

[1] http://www.mozart-oz.org/

[2] http://jocaml.inria.fr/

[3] G. Fournet and G. Gonthier: The join calculus: a language for distributed mobile programming, Springer-Verlag, 2002

[4] G. S. Itzstein and D. Kearney: Applications of Join Java, In proceedings of the Seventh Asia-Pacific Computer Systems Architechture Conference, 2002

[5] C. Hewitt, P. Bishop and R. Steiger: A Universal Modular Actor Formalism for Artificial Intelligence, IJCAI, 1973

[6] http://www.javaworld.com/javaworld/javaqa/2001-04/02-qa-0413-four.html

[7] E. Shapiro: The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 1989

[8] V. A. Saraswat and M. Rinard: Concurrent Constraint Programming, ACM Symposium on Principles of Programming Languages, 1990

[9] F. Drejhammar, C. Schulte, P. Brand and S. Haridi: Flow Java: Declarative Concurrency for Java, 2003