Explicit concurrent programming in high-level languages

Kari Kähkönen

March 19, 2009
Outline

1. Introduction
2. Join calculus
   - Overview
   - JoCaml
   - Other languages
3. Actor model
   - Overview
   - Erlang
4. Data-flow programming
   - Overview
   - Flow-Java
   - Oz
Explicit parallelism

- In languages that use explicit parallelism the programmer must explicitly 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.
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”).

Kari Kähkönen
Explicit concurrent programming in high-level languages
CHAM example

Reduction rule $A \& B \rightarrow C \& F$
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
Channels and processes

- Channels are created with a `def` binding

  ```caml
  #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

  ```caml
  #spawn echo(1) & echo(2)
  #spawn begin
  print_int 1; print_int 2; 0
  end
  ```
Channels and processes

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

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

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

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

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

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

```ocaml
#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
Join patterns define multiple channels and specifies a synchronization pattern between them:

```c
#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:

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

```c
#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.
Mutual exclusion example

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

  ```
  #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()
  ```
Many common synchronization primitives can be expressed with Join patterns

Locks:

```ocaml
#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()
```
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))
Timeouts

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

```plaintext
#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 archive this.
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

```java
final class SimpleJoinPattern {
    int A() & B() & C(int x) {
        return x;
    }
}

final class SimpleJoinThread {
    signal athread(int x) {
        ...
    }
}
```
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
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
- ActorScript
- etc.
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.
Processes

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

\[ \text{Pid2} = \text{spawn}(\text{Mod, Func, Args}) \]

- 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
In the example below, `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.

```erlang
-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.
```
[example continued from the previous slide]

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

Kari Kähkönen
Explicit concurrent programming in high-level languages
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
Registered processes

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

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

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

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

- The message buffer can be flushed followingly

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

```java
single Object s;
Object o = new Object();
s @= o;
```
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);
}
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.

```java
public static Object spawn() {
    single Object r;
    new Thread(new Spawn(r)).start();
    return r;
}
```
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...]
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;
}
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
A simple example

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

```plaintext
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
```

Kari Kähkönen

Explicit concurrent programming in high-level languages
A concurrent map function

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

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

Kari Kähkönen

Explicit concurrent programming in high-level languages
Streams

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

```plaintext
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 
end 
'in fun {Sum L} {Sum1 L 0} end end
{Browse thread {Sum thread {Generator 100} end} end}
```
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.
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}
Questions?