

## Declarative Concurrency (VRH 4)

Carlos Varela  
Rensselaer Polytechnic Institute

Adapted with permission from:  
Seif Haridi  
KTH  
Peter Van Roy  
UCL

November 26, 2007

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

1

## Review of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language, Oz)
  - Message passing with active objects (Erlang, SALSA)
  - Atomic actions on shared state (Java)
- The atomic action approach is the *most difficult*, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
  - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, else message passing if you can get away with it.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

2

## Concurrency

- Some programs are best written as a set of activities that run independently (concurrent programs)
- Concurrency is essential for interaction with the external environment
- Examples includes GUI (Graphical User Interfaces), operating systems, web services
- Also programs that are written independently but interact only when needed (client-server, peer-to-peer applications)
- This lecture is about declarative concurrency, programs with no observable nondeterminism, the result is a function
- Independent procedures that execute on their pace and may communicate through shared dataflow variables

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

3

## Overview

- Programming with threads
  - The model is augmented with threads
  - Programming techniques: stream communication, order-determining concurrency, coroutines, concurrent composition
- Lazy execution
  - demand-driven computations, lazy streams, and list comprehensions
- Soft real-time programming

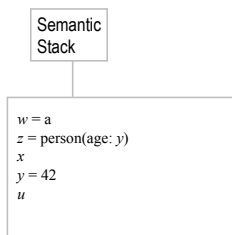
C. Varela; Adapted with permission from S. Haridi and P. Van Roy

4

## The sequential model

Statements are executed sequentially from a single semantic stack

Single-assignment store



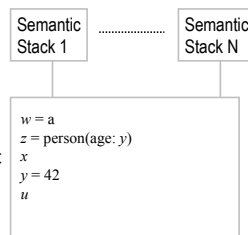
C. Varela; Adapted with permission from S. Haridi and P. Van Roy

5

## The concurrent model

Multiple semantic stacks (threads)

Single-assignment store



C. Varela; Adapted with permission from S. Haridi and P. Van Roy

6

## Concurrent declarative model

The following defines the syntax of a statement,  $\langle s \rangle$  denotes a statement

$\langle s \rangle ::=$	<code>skip</code>	<i>empty statement</i>
	<code>(x) = (y)</code>	<i>variable-variable binding</i>
	<code>(x) = (v)</code>	<i>variable-value binding</i>
	<code>(s<sub>1</sub>) (s<sub>2</sub>)</code>	<i>sequential composition</i>
	<code>local (x) in (s<sub>1</sub>) end</code>	<i>declaration</i>
	<code>proc {(x) (y<sub>1</sub>) ... (y<sub>n</sub>)} (s<sub>1</sub>) end</code>	<i>procedure introduction</i>
	<code>if (x) then (s<sub>1</sub>) else (s<sub>2</sub>) end</code>	<i>conditional</i>
	<code>{(x) (y<sub>1</sub>) ... (y<sub>n</sub>)}</code>	<i>procedure application</i>
	<code>case (x) of (pattern) then (s<sub>1</sub>) else (s<sub>2</sub>) end</code>	<i>pattern matching</i>
	<code>thread (s<sub>1</sub>) end</code>	<i>thread creation</i>

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

7

## The concurrent model

Top of Stack, Thread  $i \rightarrow$   $\begin{array}{|c|} \hline \text{ST} \\ \hline \text{thread } \langle s_1 \rangle \text{ end, } E \\ \hline \end{array}$

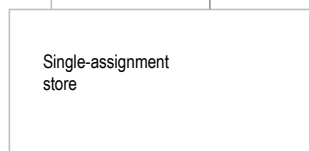


C. Varela; Adapted with permission from S. Haridi and P. Van Roy

8

## The concurrent model

Top of Stack, Thread  $i \rightarrow$   $\begin{array}{|c|} \hline \text{ST} \\ \hline \end{array}$       $\begin{array}{|c|} \hline \langle s_1 \rangle, E \\ \hline \end{array}$



C. Varela; Adapted with permission from S. Haridi and P. Van Roy

9

## Basic concepts

- The model allows multiple statements to execute "at the same time".
- Imagine that these threads really execute in parallel, each has its own processor, but share the same memory
- Reading and writing different variables can be done simultaneously by different threads, as well as reading the same variable
- Writing the same variable is done sequentially
- The above view is in fact equivalent to an *interleaving execution*: a totally ordered sequence of computation steps, where threads take turns doing one or more steps in sequence

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

10

## Causal order

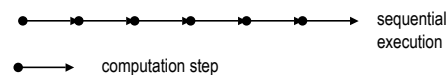
- In a sequential program all execution states are totally ordered
- In a concurrent program all execution states of a given thread are totally ordered
- The execution state of the concurrent program as a whole is partially ordered

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

11

## Total order

- In a sequential program all execution states are totally ordered

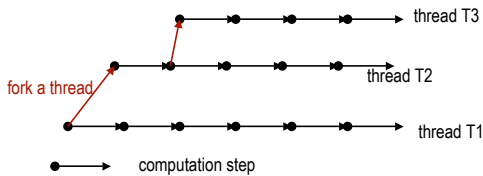


C. Varela; Adapted with permission from S. Haridi and P. Van Roy

12

## Causal order in the declarative model

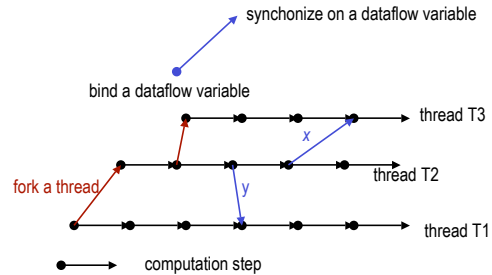
- In a concurrent program all execution states of a given thread are totally ordered
- The execution state of the concurrent program is partially ordered



C. Varela; Adapted with permission from S. Haridi and P. Van Roy

13

## Causal order in the declarative model



C. Varela; Adapted with permission from S. Haridi and P. Van Roy

14

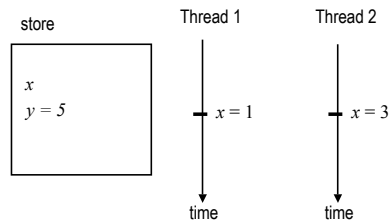
## Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is concurrent access to shared state

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

15

## Example of nondeterminism



The thread that binds  $x$  first will continue, the other thread will raise an exception

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

16

## Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is concurrent access to shared state
- In the concurrent declarative model when there is only one binder for each dataflow variable, the nondeterminism is not observable on the store (i.e. the store develops to the same final results)
- This means for correctness we can ignore the concurrency

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

17

## Scheduling

- The choice of which thread to execute next and for how long is done by a part of the system called the *scheduler*
- A thread is *runnable* if its next statement to execute is not blocked on a dataflow variable, otherwise the thread is *suspended*
- A scheduler is fair if it does not starve a runnable thread, i.e. all runnable threads eventually execute
- Fair scheduling makes it easy to reason about programs and program composition
- Otherwise some correct program (in isolation) may never get processing time when composed with other programs

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

18

## The semantics

- In the sequential model we had:  
( $ST, \sigma$ )

$ST$  is a stack of semantic statements  
 $\sigma$  is the single assignment store

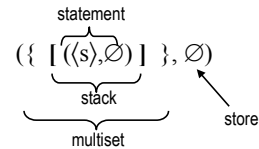
- In the concurrent model we have:  
( $MST, \sigma$ )

$MST$  is a (multi)set of stacks of semantic statements  
 $\sigma$  is the single assignment store

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

19

## The initial execution state



C. Varela; Adapted with permission from S. Haridi and P. Van Roy

20

## Execution (the scheduler)

- At each step, one **runnable** semantic stack is *selected* from  $MST$  (the multiset of stacks), call it  $ST$ , s.t.  
 $MST = ST \cup MST'$
- Assume the current store is  $\sigma$ , one computation step is done that transforms  $ST$  to  $ST'$  and  $\sigma$  to  $\sigma'$
- The total computation state is transformed from  $(MST, \sigma)$  to  $(ST' \cup MST', \sigma')$
- Which stack is selected, and how many steps are taken is the task of the scheduler, a good scheduler should be fair, i.e., each runnable 'thread' will eventually be selected
- The computation stops when there are no runnable stacks

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

21

## Example of runnable threads

```

proc {Loop P N}
  if N > 0 then
    {P} {Loop P N-1}
  else skip end
end
thread {Loop
  proc {$} {Show 1} end
  1000}
end
thread {Loop
  proc {$} {Show 2} end
  1000}
end

```

- This program will interleave the execution of two threads, one printing 1, and the other printing 2
- We assume a fair scheduler

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

22

## Dataflow computation

- Threads suspend on data unavailability in dataflow variables
- The **{Delay X}** primitive makes the thread suspends for X milliseconds, after that, the thread is runnable

```

declare X
{Browse X}
local Y in
  thread {Delay 1000} Y = 10*10 end
  X = Y + 100*100
end

```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

23

## Illustrating dataflow computation

```

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

```

- Enter incrementally the values of  $X_0$  to  $X_3$
- When  $X_0$  is bound the thread will compute  $Y_0 = X_0 + 1$ , and will suspend again until  $X_1$  is bound

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

24

## Concurrent Map

```

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

- This will fork a thread for each individual element in the input list
- Each thread will run only if both the element X and the procedure F is known

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

25

## Concurrent Map Function

```

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

- What this looks like in the kernel language:

```

proc {Map Xs F Rs}
  case Xs
  of nil then Rs = nil
  [] X|Xr then R Rr in
    Rs = R|Rr
    thread R = {F X} end
    {Map Xr F Rr}
  end
end
    
```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

26

## How does it work?

- If we enter the following statements:  

```
declare F X Y Z
{Browse thread {Map X F} end}
```
- A thread executing Map is created.
- It will suspend immediately in the case-statement because X is unbound.
- If we thereafter enter the following statements:  

```
X = 1|2|Y
fun {F X} X*X end
```
- The main thread will traverse the list creating two threads for the first two arguments of the list

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

27

## How does it work?

- The main thread will traverse the list creating two threads for the first two arguments of the list:  

```
thread {F 1} end, and thread {F 2} end,
```

After entering:

```
Y = 3|Z
Z = nil
```

the program will complete the computation of the main thread and the newly created thread `thread {F 3} end`, resulting in the final list `[1 4 9]`.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

28

## Simple concurrency with dataflow

- Declarative programs can be easily made concurrent
- Just use the thread statement where concurrency is needed

```

fun {Fib X}
  if X=<2 then 1
  else
    thread {Fib X-1} end + {Fib X-2}
  end
end
    
```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

29

## Understanding why

```

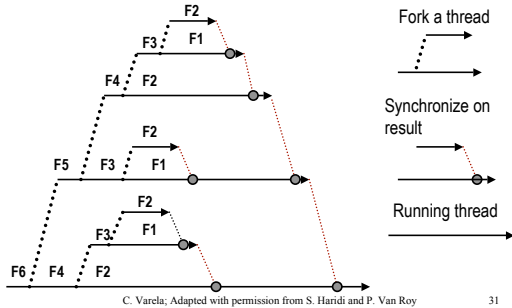
fun {Fib X}
  if X=<2 then 1
  else F1 F2 in
    F1 = thread {Fib X-1} end
    F2 = {Fib X-2}
    F1 + F2
  end
end
    
```

Dataflow dependency

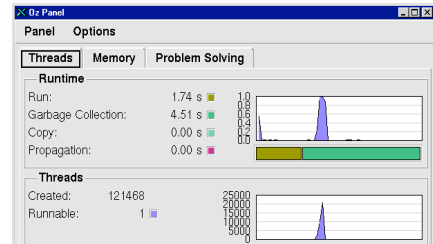
C. Varela; Adapted with permission from S. Haridi and P. Van Roy

30

## Execution of {Fib 6}

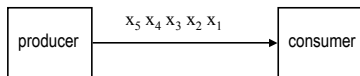


## Threads and Garbage Collection



## Streams

- A stream is a sequence of messages
- A stream is a First-In First-Out (FIFO) channel
- The producer augments the stream with new messages, and the consumer reads the messages, one by one.



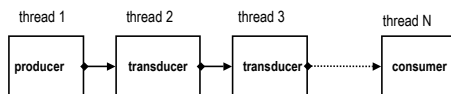
## Stream Communication I

- The data-flow property of Oz easily enables writing threads that communicate through streams in a producer-consumer pattern.
- A stream is a list that is created incrementally by one thread (the producer) and subsequently consumed by one or more threads (the consumers).
- The consumers consume the same elements of the stream.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 34

## Stream Communication II

- **Producer**, produces incrementally the elements
- **Transducer(s)**, transform(s) the elements of the stream
- **Consumer**, accumulates the results



## Stream communication patterns

- The producer, transducers, and the consumer can, in general, be described by certain program patterns
- We show various patterns

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 36

## Producer

```

fun {Producer State}
  if {More State} then
    X = {Produce State} in
    X | {Producer {Transform State}}
  else nil end
end

```

- The definition of *More*, *Produce*, and *Transform* is problem dependent
- State could be multiple arguments
- The above definition is not a complete program!

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

37

## Example Producer

```

fun {Generate N Limit}
  if N=<Limit then
    N | {Generate N+1 Limit}
  else nil end
end

fun {Producer State}
  if {More State} then
    X = {Produce State} in
    X | {Producer {Transform State}}
  else nil end
end

```

- The State is the two arguments N and Limit
- The predicate *More* is the condition  $N \leq \text{Limit}$
- The predicate *Produce* is the identity function on N
- The Transform function  $(N, \text{Limit}) \Rightarrow (N+1, \text{Limit})$

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

38

## Consumer Pattern

```

fun {Consumer State InStream}
  case InStream
  of nil then {Final State}
  [] X | RestInStream then
    NextState = {Consume X State} in
    {Consumer NextState RestInStream}
  end
end

```

The consumer suspends until InStream is either a cons or a nil

- *Final* and *Consume* are problem dependent

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

39

## Example Consumer

```

fun {Sum A Xs}
  case Xs
  of nil then A
  [] X|Xr then {Sum A+X Xr}
  end
end

fun {Consumer State InStream}
  case InStream
  of nil then {Final State}
  [] X | RestInStream then
    NextState = {Consume X State} in
    {Consumer NextState RestInStream}
  end
end

```

- The State is A
- *Final* is just the identity function on State
- *Consume* takes X and State  $\Rightarrow X + \text{State}$

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

40

## Transducer Pattern 1

```

fun {Transducer State InStream}
  case InStream
  of nil then nil
  [] X | RestInStream then
    NextState#TX = {Transform X State}
    TX | {Transducer NextState RestInStream}
  end
end

```

- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

41

## Transducer Pattern 2

```

fun {Transducer State InStream}
  case InStream
  of nil then nil
  [] X | RestInStream then
    if {Test X#State} then
      NextState#TX = {Transform X State}
      TX | {Consumer NextState RestInStream}
    else {Consumer NextState RestInStream} end
  end
end

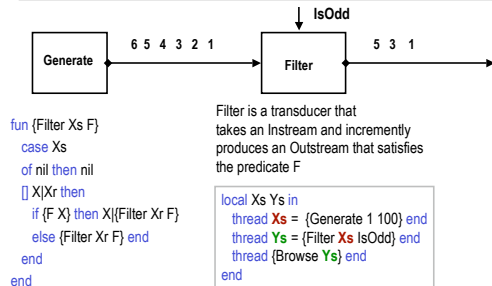
```

- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

42

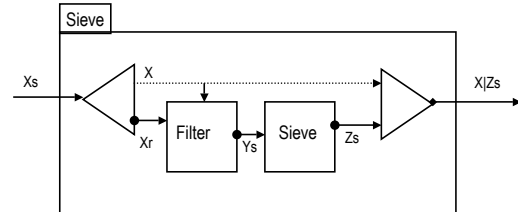
## Example Transducer



C. Varela; Adapted with permission from S. Haridi and P. Van Roy 43

## Larger example: The sieve of Eratosthenes

- Produces prime numbers
- It takes a stream 2...N, peels off 2 from the rest of the stream
- Delivers the rest to the next sieve



C. Varela; Adapted with permission from S. Haridi and P. Van Roy 44

## Sieve

```

fun {Sieve Xs}
case Xs
of nil then nil
[] X|Xr then Ys in
thread Ys = {Filter Xr fun {$ Y} Y mod X != 0 end} end
X | {Sieve Ys}
end
end
    
```

- The program forks a filter thread on each sieve call

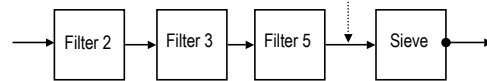
C. Varela; Adapted with permission from S. Haridi and P. Van Roy 45

## Example call

```

local Xs Ys in
thread Xs = {Generate 2 100000} end
thread Ys = {Sieve Xs} end
thread for Y in Ys do {Show Y} end end
end
    
```

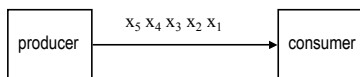
7 | 11 | ...



C. Varela; Adapted with permission from S. Haridi and P. Van Roy 46

## Limitation of eager stream processing Streams

- The producer might be much faster than the consumer
- This will produce a large intermediate stream that requires potentially unbounded memory storage



C. Varela; Adapted with permission from S. Haridi and P. Van Roy 47

## Solutions

There are three alternatives:

1. Play with the speed of the different threads, i.e. play with the scheduler to make the producer slower
  2. Create a bounded buffer, say of size N, so that the producer waits automatically when the buffer is full
  3. Use demand-driven approach, where the consumer activates the producer when it needs a new element (**lazy evaluation**)
- The last two approaches introduce the notion of flow-control between concurrent activities (very common)

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 48

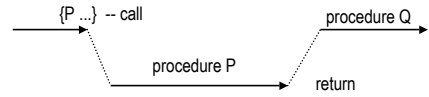


## Coroutines I

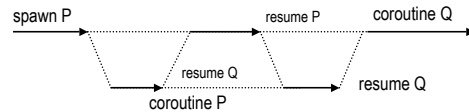
- Languages that do not support concurrent threads might instead support a notion called **corouting**
- A coroutine is a nonpreemptive thread (sequence of instructions), there is no scheduler
- Switching between threads is the programmer's responsibility

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 49

## Coroutines II, Comparison

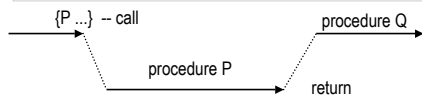


**Procedures:** one sequence of instructions, program transfers explicitly when terminated it returns to the caller

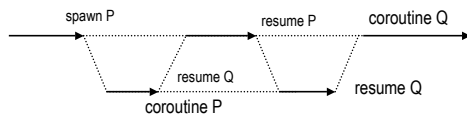


C. Varela; Adapted with permission from S. Haridi and P. Van Roy 50

## Coroutines II, Comparison



**Coroutines:** New sequences of instructions, programs explicitly do all the scheduling, by spawn, suspend and resume



C. Varela; Adapted with permission from S. Haridi and P. Van Roy 51

## Time

- In concurrent computation one would like to handle time
- `proc {Time.delay T}` – The running thread suspends for T milliseconds
- `proc {Time.alarm T U}` – Immediately creates its own thread, and binds U to `unit` after T milliseconds

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 52

## Example

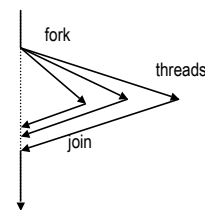
```
local
proc {Ping N}
  for I in 1..N do
    {Delay 500} {Browse ping}
  end
  {Browse 'ping terminate'}
end
proc {Pong N}
  for I in 1..N do
    {Delay 600} {Browse pong}
  end
  {Browse 'pong terminate'}
end
in ... end
```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 53

```
local
...
in
  {Browse 'game started'}
  thread {Ping 1000} end
  thread {Pong 1000} end
end
```

## Concurrent control abstraction

- We have seen how threads are forked by 'thread ... end'
- A natural question to ask is: how can we join threads?



C. Varela; Adapted with permission from S. Haridi and P. Van Roy 54

## Termination detection

- This is a special case of detecting *termination of multiple threads*, and making another thread wait on that event.
- The general scheme is quite easy because of dataflow variables:
 

```

thread (S1) X1 = unit end
thread (S2) X2 = X1 end
...
thread (Sn) Xn = Xn-1 end
{Wait Xn}
% Continue main thread
      
```
- When all threads terminate the variables  $X_1 \dots X_n$  will be merged together labeling a single box that contains the value **unit**.
- {Wait  $X_n$ } suspends the main thread until  $X_n$  is bound.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 55

## Concurrent Composition

`conc S1 [] S2 [] ... [] Sn end`

```

{Conc [ proc{$} S1 end
      proc{$} S2 end
      ...
      proc{$} Sn end ] }
      
```

- Takes a single argument that is a list of nullary procedures.
- When it is executed, the procedures are forked concurrently. The next statement is executed only when all procedures in the list terminate.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 56

## Conc

```

local
proc {Conc1 Ps I O}
case Ps of P|Pr then
M in
thread {P} M = I end
{Conc1 Pr M O}
[] nil then O = I
end
end
in
proc {Conc Ps}
X in {Conc1 Ps unit X}
{Wait X}
end
end
      
```

This abstraction takes a list of zero-argument procedures and terminate after all these threads have terminated

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 57

## Example

```

local
proc {Ping N}
for I in 1..N do
{Delay 500} {Browse ping}
end
{Browse 'ping terminate'}
end
proc {Pong N}
for I in 1..N do
{Delay 600} {Browse pong}
end
{Browse 'pong terminate'}
end
in ... end
      
```

```

local
....
in
{Browse 'game started'}
{Conc
[ proc {$} {Ping 1000} end
  proc {$} {Pong 1000} end ]}
{Browse 'game terminated'}
end
      
```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 58

## Futures

- A **future** is a read-only capability of a single-assignment variable. For example to create a future of the variable  $X$  we perform the operation `!!X` to create a future  $Y$ :  $Y = !!X$
- A thread trying to use the value of a future, e.g. using  $Y$ , will suspend until the variable of the future, e.g.  $X$ , gets bound.
- One way to execute a procedure lazily, i.e. in a demand-driven manner, is to use the operation `{ByNeed +P ?F}`.
- `ByNeed` takes a zero-argument function  $F$ , and returns a future  $F$ . When a thread tries to access the value of  $F$ , the function `{P}` is called, and its result is bound to  $F$ .
- This allows us to perform demand-driven computations in a straightforward manner.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 59

## Example

```

declare Y
{ByNeed fun {$} 1 end Y}
{Browse Y}
      
```

- we will observe that  $Y$  becomes a future, i.e. we will see `Y<Future>` in the Browser.
- If we try to access the value of  $Y$ , it will get bound to 1.
- One way to access  $Y$  is by perform the operation `{Wait Y}` which triggers the producing procedure.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy 60

## Thread Priority and Real Time

- Try to run the program using the following statement:
  - {Sum 0 **thread** {Generate 0 100000000} **end**}
- Switch on the panel and observe the memory behavior of the program.
- You will quickly notice that this program does not behave well.
- The reason has to do with the asynchronous message passing. If the producer sends messages i.e. create new elements in the stream, in a faster rate than the consumer can consume, increasingly more buffering will be needed until the system starts to break down.
- One possible solution is to control experimentally the rate of thread execution so that the consumers get a larger time-slice than the producers do.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

61

## Priorities

- There are three priority levels:
  - *high*,
  - *medium*, and
  - *low* (the default)
- A priority level determines how often a runnable thread is allocated a time slice.
- In Oz, a high priority thread cannot starve a low priority one. Priority determines only how large piece of the processor-cake a thread can get.
- Each thread has a unique name. To get the name of the current thread the procedure `Thread.this/i` is called.
- Having a reference to a thread, by using its name, enables operations on threads such as:
  - terminating a thread, or
  - raising an exception in a thread.
- Thread operations are defined the standard module `Thread`.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

62

## Thread priority and thread control

```
fun {Thread.state T}          %% returns thread state
proc {Thread.injectException T E} %% exception E injected into thread
fun {Thread.this}           %% returns 1st class reference to thread
proc {Thread.setPriority T P} %% P is high, medium or low
proc {Thread.setThisPriority P} %% as above on current thread

fun {Property.get priorities} %% get priority ratios
proc {Property.put priorities (high:H medium:M)}
```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

63

## Thread Priorities

- Oz has three priority levels. The system procedure
  - {Property.put priorities p (medium:Y high:X)}
- Sets the processor-time ratio to X:1 between high-priority threads and medium-priority thread.
- It also sets the processor-time ratio to Y:1 between medium-priority threads and low-priority threads. X and Y are integers.
  - Example:
    - {Property.put priorities p (high:10 medium:10)}
- Now let us make our producer-consumer program work. We give the producer low priority, and the consumer high. We also set the priority ratios to 10:1 and 10:1.

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

64

## The program with priorities

```
local L in
  {Property.put priorities p (high:10 medium:10)}
  thread
    {Thread.setThisPriority low}
    L = {Generate 0 100000000}
  end
  thread
    {Thread.setThisPriority high}
    {Sum 0 L}
  end
end
```

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

65

## Exercises

89. SALSA asynchronous message passing enables to tag messages with properties: *priority*, *delay*, and *waitfor*. Compare these mechanisms with Oz thread priorities, time delays and alarms, and futures.
90. How do SALSA tokens relate to Oz dataflow variables and futures?
91. What is the difference between multiple thread termination detection in Oz and join blocks in SALSA?
92. \*VRH Exercise 4.11.3 (page 339)  
Compare the sequential and concurrent execution performance of equivalent SALSA programs.
93. \*VRH Exercise 4.11.5 (page 339)

C. Varela; Adapted with permission from S. Haridi and P. Van Roy

66