Programming Language Concepts, cs2104
Lecture 10 (2003-10-10)

Seif Haridi
Department of Computer Science, NUS
haridi@comp.nus.edu.sg

Reading Suggestions

- Chapter 4
  - Sections 4.1-4.6 [careful]
- And of course the handouts!

Organizational

- Assignment 4 is out there
- Check your marks for midterm?
- Next week no lecture
- Next week tutorials as usual
- The week after no tutorials

Concurrency

Concurrency

- First: declarative concurrency
- What is concurrency?
- How to make a program concurrent?
- How do concurrent programs execute?

The World Is Concurrent!

- Concurrent programs
  several activities execute
  simultaneously (concurrently)
- Most of the software you use is concurrent
  - operating system: IO, user interaction, many processes, …
  - web browser, Email client, Email server, …
  - telephony switches handling many calls
  - …

Why Should We Care?

- Software must be concurrent…

  … for many application areas
- Concurrency can be helpful for constructing programs
  - organize programs into independent parts
  - concurrency allows to make them independent with respect to how the execute
  - essential: how do concurrent programs interact?
- Concurrent programs can run faster on parallel machines (including clusters)

Concurrent Programming Is Difficult…

- This is the traditional belief
- The truth is: concurrency is very difficult…

  … if used with inappropriate tools and programming languages
- In particular troublesome: state and concurrency
  (see discussion end of Chapter 1)

Concurrent Programming Is Easy…

- Oz (as well as Erlang) has been designed to be very good at concurrency…

- Essential for concurrent programming here
  - data-flow variables

    very simple interaction between concurrent programs, mostly automatic
  - light-weight threads

Declarative Concurrent Programming

- What stays the same
  - the result of your program
  - concurrency does not change the result

- What changes
  - programs can compute incrementally
  - incremental input… (such as reading from a network connection)

    …is processed incrementally
  - the fun: much greater!

Threads

The sequential model

Statements are executed sequentially from a single semantic stack

Semantic Stack

Single-assignment store

w = a
z = person(age: *y*)
x
y = 42
u

The concurrent model

Multiple semantic stacks (threads)

Semantic Stack 1 .... Semantic Stack N

Single-assignment store

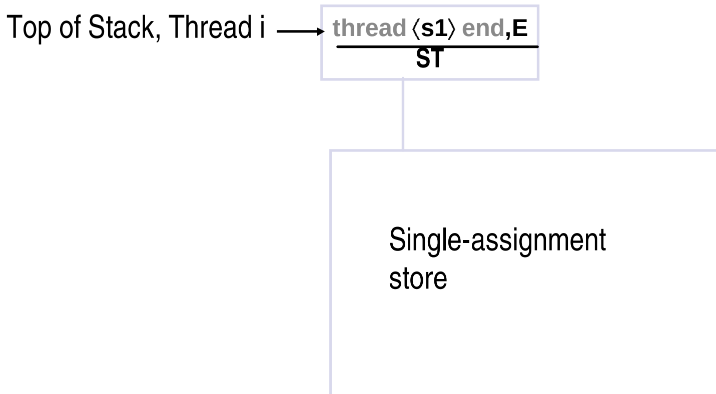
w = a
z = person(age: y)
x
y = 42
u

Concurrent declarative model

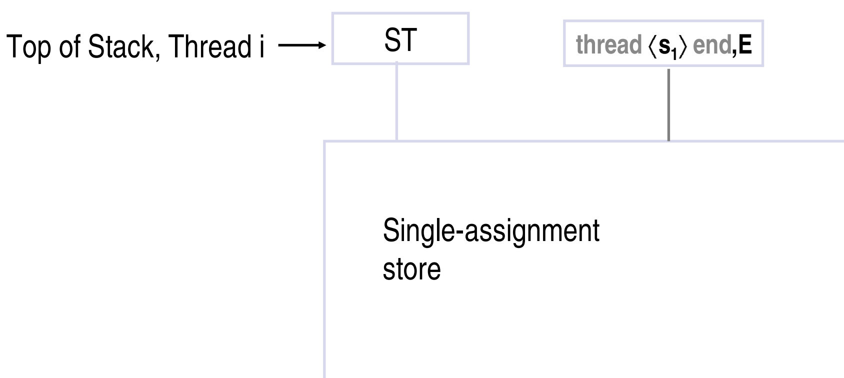
The following defines the syntax of a statement, ⟨s⟩ denotes a statement

⟨s⟩ ::=	skip	*empty statement*
|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
|	⟨x⟩ = ⟨v⟩	*variable-value binding*
|	$\langle s_1 \rangle$ $\langle s_2 \rangle$	*sequential composition*
|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
|	proc {⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ } $\langle s_1 \rangle$ end	*procedure introduction*
|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
|	{ ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ }	*procedure application*
|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*
|	**thread $\langle s_1 \rangle$ end**	***thread creation***

The concurrent model



The concurrent model



Data driven computation

- Threads suspend of data availability 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
```


Concurrency Is Transparent

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

Cheap concurrency and dataflow

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

Cheap concurrency and dataflow

```
fun {Fib X}
   if X==0 then 0
   elseif X==1 then 1
   else
      thread {Fib X-1} end + {Fib X-2}
   end
end
```

Producer ⇔ Consumer

```
thread X={Produce} end
thread {Consume X} end
```

- Typically, what is produced will be put on a list that never ends (without `nil`)
  - stream
- Consumer consumes as soon as producer produces

Example: Producer ⇔ Consumer

```
fun {Produce N}
   N|{Produce N+1}
end
proc {Consume Xs}
   case Xs of X|Xr then
      if X mod 1000 == 0 then
         {Browse X}
      end
      {Consume Xr}
   end
end
```

Stream Transducer

```
thread Xs={Produce}        end
thread Ys={Transduce Xs}   end
thread {Consume Ys}        end
```

- Transducer
  - reads input stream
  - computes output stream
- Can be: filtering, mapping, …

Concurrent Streams

- Often used for simulation
  - analog circuits
  - digital circuits
- Lab assignment 4
  - streams used for simulation of analog circuits
  - simple circuits
  - **lazy** streams

Summary

- Threads
  - suspend and resume automatically
  - controlled by variables
  - reminder: **data-flow variables**
  - cheap
  - execute fairly according to time-slice
- Pattern
  - producer ⇔ transducer ⇔ consumer

Demand Driven Execution

How to Control Producers?

- Producer should not produce more than needed
- Make consumer the stream producer
  - consumer produces skeleton, producer fills skeleton
  - difficult
- **Use lazy streams: producer runs on request**

Demand-driven Execution

- Let computations drive other computations
  - producer driven by consumer/transducer
  - module loader by thread needing module
- Variables control “demand” or “need”
  - variable needed: thread suspends on variable
  - by-need trigger:
    - variable
    - nullary function describing value to be computed
  - execution by newly created thread

Needed Variables

- Idea: start execution, when value for variable needed

  short: **variable needed**

- Value for variable needed…

  …a thread suspends on variable!

Triggers

- By-need triggers
  - a variable X
  - a zero-argument function F

- Trigger creation

```
X={ByNeed F}
```

The By-Need Protocol

- Suppose `(X,F)` is a by-need trigger

- If `X` is needed,
  - execute `thread X={F} end`
  - delete trigger, `X` becomes a normal variable

Lazy Functions

```
fun lazy {Produce N}
   N|{Produce N+1}
end
```

abbreviates

```
fun {Produce N}
   {ByNeed fun {$} N|{Produce N+1} end}
end
```

Summary

- Demand-driven execution
  - execute computation, if variable needed
  - need is suspension by a thread
  - requested computation is run in new thread
- By-Need triggers
- Lazy functions

Thread Semantics

Semantics for Threads

- We insist on ***interleaving*** semantics
  - model: only one thread executes at a time
  - implementation: might execute several threads in parallel, however must execute as if one thread at a time
- Important property: ***monotonicity***
  - if a thread becomes runnable:

    …it stays runnable

    …doesn't matter when it is actually run

Monotonicity

- Example:

```
thread
   if B then … else … end
end
```

- When B is bound, thread will eventually run
- When B is bound, the value of B is fixed
  - value of B independent of when thread executes

Monotonicity: Simplicity

- Result is scheduling independent
  - unless attempt to put inconsistent information to store
  - example for non-determinism

```
thread X=1 end
thread X=2 end
```

  which value for X?

- Different with explicit mutable state (JAVA)
  - if variable values changed over time, result would depend on order in which threads run

Dependencies

- Suspension and resumption driven by variable bindings
- Progress, only if for each variable actually value supplied
- Typical error-scenario: deadlock
  - thread depends on X, supposed to bind Y
  - thread depends on Y, supposed to bind X

Extend Abstract Machine

- Extend to execute multiple threads
  - shared store: all threads share common store
  - semantic stack: corresponds to a thread
  - thread creation: create new semantic stack
- Orthogonal: scheduling policy
  - scheduling policy: which thread to execute?
  - consider only non-suspended threads!

Abstract Machine Concepts…

- Single-assignment store
- Environment
- Semantic statement
- Execution state
- Computation

Abstract Machine

- Performs a computation
- *Computation* is sequence of execution states
- *Execution state*
  - stack of semantic statements
  - single assignment store
- *Semantic statement*
  - statement
  - environment
- *Environment* maps variable identifiers to store entities

Single Assignment Store

- Single assignment store σ
  - set of store variables
  - partitioned into
    - sets of variables that are equal but unbound
    - variables bound to value
- Example store $\{x_1, x_2{=}x_3, x_4{=}a|x_2\}$
  - x_1 unbound
  - x_2, x_3 equal and unbound
  - x_4 bound to partial value $a|x_2$

Environment

- Environment E
  - maps variable identifiers to entities in store σ
  - written as set of pairs $X \rightarrow x$
    - variable identifier X
    - store variable x
- Example environment $\{ X \rightarrow x, Y \rightarrow y \}$
  - maps identifier X to store variable x
  - maps identifier Y to store variable y

Environment and Store

- Given: environment E, store σ
- Looking up value for variable identifier X:
  - find store variable in environment $E(X)$
  - take value from σ for $E(X)$
- Example:

  $\sigma=\{x_1, x_2=x_3, x_4=a|x_2\}$ $E = \{ X \rightarrow x_1, Y \rightarrow x_4 \}$
  - $E(X) = x_1$ and no information in σ on x_1
  - $E(Y) = x_4$ and σ binds x_4 to $a|x_2$

Environment Adjunction

- Given: Environment E

  $E + \{\langle x \rangle_1 \rightarrow x_1, \ldots, \langle x \rangle_n \rightarrow x_n\}$

  is new environment E' with mappings added:
  - always take store entity from new mappings
  - might overwrite old mappings

Semantic Statements

- To actually execute statement:
  - environment to map identifiers
    - modified with execution of each statement
    - each statement has its own environment
  - store to find values
    - all statements modify same store
    - single store
- Semantic statement $(\langle s \rangle, E)$
  - pair of (statement, environment)

Semantic Stack

- Execution maintains stack of semantic statements *ST*

  $[(\langle s \rangle_1, E_1), \ldots, (\langle s \rangle_n, E_n)]$

  - always topmost statement $(\langle s \rangle_1, E_1)$ executes first
  - rest of stack: what needs to be done

Semantic Stack States

- Semantic stack can be in run-time states
  - *terminated* stack is empty
  - *runnable* can do execution step
  - *suspended* stack not empty, no execution step possible
- Statements
  - *non-suspending* can always execute
  - *suspending* need values from store dataflow behavior

Summary

- Single assignment store — σ
- Environments — *E*
  - adjunction — *E* + {…}
- Semantic statements — (⟨s⟩, *E*)
- Semantic stacks — [(⟨s⟩, *E*) …]
- Execution state — (*ST*, σ)
- Program execution
  - runnable, terminated, suspended
- Statements
  - suspending, non-suspending

Executing `if`

- Semantic statement is
  (`if` ⟨x⟩ `then` $\langle s\rangle_1$ `else` $\langle s\rangle_2$ `end`, *E*)
- If activation condition "⟨x⟩ bound" true
  - if *E*(⟨x⟩) bound to `true` — push $\langle s\rangle_1$
  - if *E*(⟨x⟩) bound to `false` — push $\langle s\rangle_2$
  - otherwise, raise error
- Otherwise, suspend…

Procedure Call

- Semantic statement is

  $(\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}, E)$

  where
  - $E(\langle x\rangle)$ is to be called
  - $\langle y\rangle_1, \ldots, \langle y\rangle_n$ are *actual parameters*

- Suspending statement, suspension condition
  - $E(\langle x\rangle)$ is determined

Summary

- Semantic statement executes by
  - popping itself — always
  - creating environment — `local`
  - manipulating store — `local, =`
  - pushing new statements — `local, if` sequential composition
- Semantic statement can suspend
  - activation condition — `if, {…}, case`
  - read store

Multiple Semantic Stacks

- Abstract machine has multiple semantic stacks
  - each semantic stack represents one thread
- Number of semantic stacks change over time
  - increase: new threads created
  - decrease: threads terminate

Multisets

- Collection of semantic stacks called
  multiset of semantic stacks
- Multiset: like a set, but maintains multiplicity
  - ordinary set: element can be contained at most once
    {1,2,3}
  - multiset: element can be contained many times
    {1,1,1,2,2,3}
    different from {1,1,2,3,3}
  - just think of: bag, collection, bunch of something
  - same thread is allowed to occur more than once

Execution State

(Multiset of semantic stacks, store)

($\{ ST_1, \ldots, ST_n \}$, σ)

- we write multisets with normal set parentheses { and }

Execution State

(Multiset of semantic stacks, store)

($\underbrace{\{ ST_1, \ldots, ST_n \}}_{MST}$, σ)

- Multisets of stacks are denoted *MST*

Initial Execution State

- Given statement $\langle s \rangle$, start execution as before with empty environment, empty store and just one thread

$$(\{ [(\langle s \rangle, \emptyset)] \}, \emptyset)$$

Initial Execution State

- Given statement $\langle s \rangle$, start execution as before with empty environment, empty store and just one thread

$$(\{ [(\langle s \rangle, \emptyset)] \}, \emptyset)$$

execution state

Initial Execution State

- Given statement ⟨s⟩, start execution as before with empty environment, empty store and just one thread

$$(\ \{\ [\ (\langle s\rangle, \emptyset)\ ]\ \}, \emptyset\ )$$

store (empty)

Initial Execution State

- Given statement ⟨s⟩, start execution as before with empty environment, empty store and just one thread

$$(\ \{\ [\ (\langle s\rangle, \emptyset)\ ]\ \}, \emptyset\ )$$

multiset of threads (just one)

Initial Execution State

- Given statement ⟨s⟩, start execution as before with empty environment, empty store and just one thread

$$(\{ [(\langle s \rangle, \emptyset)] \}, \emptyset)$$

thread
(just one)

Initial Execution State

- Given statement ⟨s⟩, start execution as before with empty environment, empty store and just one thread

$$(\{ [(\langle s \rangle, \emptyset)] \}, \emptyset)$$

semantic
statement

Initial Execution State

- Given statement ⟨s⟩, start execution as before with empty environment, empty store and just one thread

$$(\{ [(\langle s \rangle, \emptyset)] \}, \emptyset)$$

statement (initial)

Initial Execution State

- Given statement ⟨s⟩, start execution as before with empty environment, empty store and just one thread

$$(\{ [(\langle s \rangle, \emptyset)] \}, \emptyset)$$

empty environment

Execution

- Execution steps

  $(MST_1, \sigma_1) \rightarrow (MST_2, \sigma_2) \rightarrow \ldots$

- At each step
  - select runnable semantic stack ST_i from MST_i
  - execute topmost semantic statement of ST_i resulting in ST'_i
  - continue with threads $MST_{i+1} = \{ST'_i\} \cup (MST_i - \{ST_i\})$

Execution

- Execution steps

  $(MST_1, \sigma_1) \rightarrow (MST_2, \sigma_2) \rightarrow \ldots$

- At each step
  - select **runnable** semantic stack ST_i from MST_i
  - execute topmost semantic statement of ST_i resulting in ST'_i
  - continue with threads $MST_{i+1} = \{ST'_i\} \cup (MST_i - \{ST_i\})$

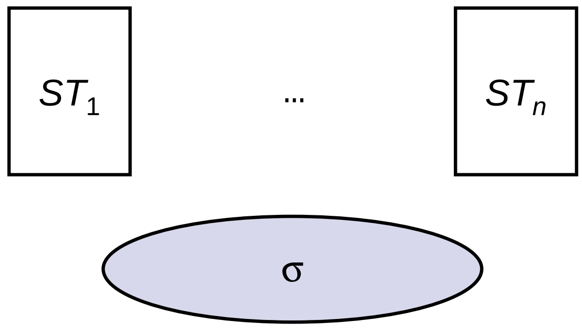
Statements

⟨s⟩ ::= **skip**
  | ⟨x⟩ = ⟨y⟩
  | ⟨x⟩ = ⟨v⟩
  | $\langle s\rangle_1$ $\langle s\rangle_2$
  | **local** ⟨x⟩ **in** ⟨s⟩ **end**
  | **if** ⟨x⟩ **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**
  | {⟨x⟩ $\langle y\rangle_1$ … $\langle y\rangle_n$}

Statements with Thread Creation

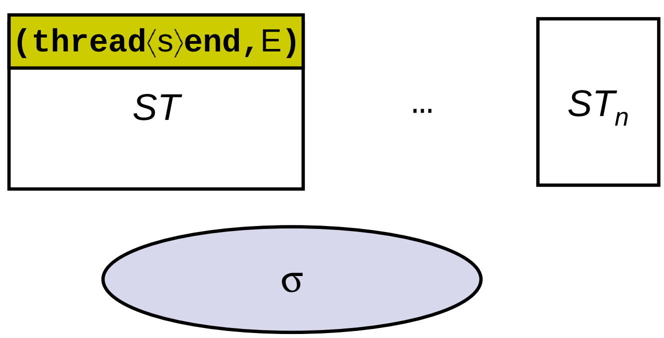
⟨s⟩ ::= **skip**
  | ⟨x⟩ = ⟨y⟩
  | ⟨x⟩ = ⟨v⟩
  | $\langle s\rangle_1$ $\langle s\rangle_2$
  | **local** ⟨x⟩ **in** ⟨s⟩ **end**
  | **if** ⟨x⟩ **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**
  | {⟨x⟩ $\langle y\rangle_1$ … $\langle y\rangle_n$}
  | **thread** ⟨s⟩ **end**

Sketch of Computation



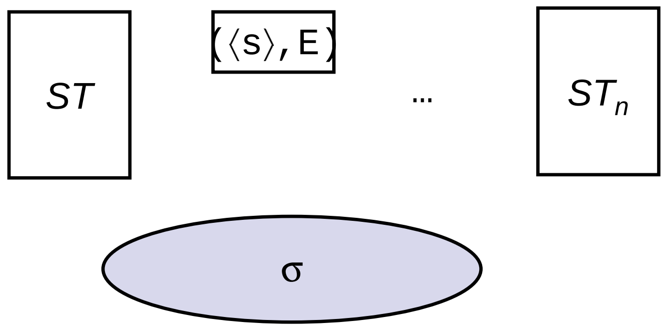
- Multiple threads sharing store

Sketch of Computation



- Thread creation statement…

Sketch of Computation



- …new semantic stack running ⟨s⟩

Example

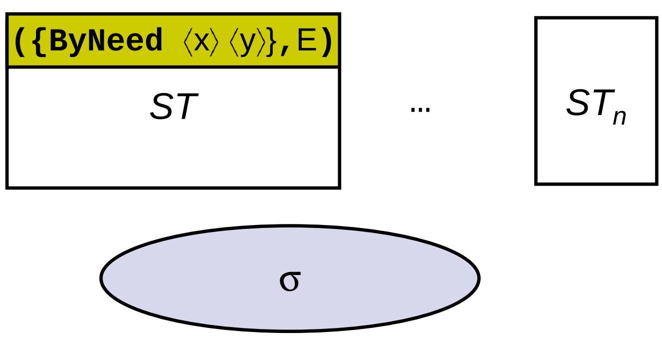
```
local B X in
   thread
      if B then X=1 else X=2 end
   end
   B=true
end
```

- see it at tutorial…

Statements with Thread Creation

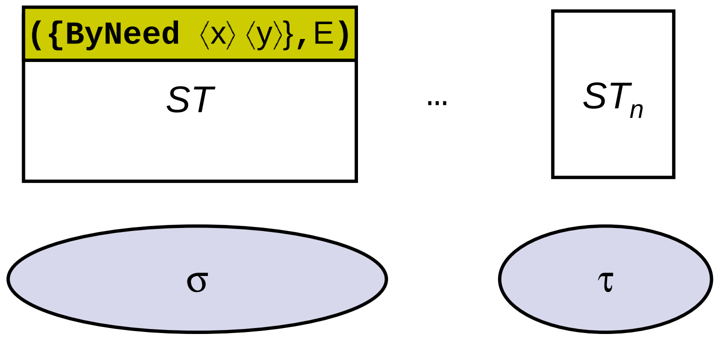
⟨s⟩ ::= **skip**
- | ⟨x⟩ = ⟨y⟩
- | ⟨x⟩ = ⟨v⟩
- | $\langle s\rangle_1$ $\langle s\rangle_2$
- | **local** ⟨x⟩ **in** ⟨s⟩ **end**
- | **if** ⟨x⟩ **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**
- | {⟨x⟩ $\langle y\rangle_1$ … $\langle y\rangle_n$}
- | **thread** ⟨s⟩ **end**
- | **{ByNeed ⟨x⟩ ⟨y⟩}**

Sketch of Computation



- Thread creation statement…

Sketch of Computation



- **The store is σ the variable-store + τ the trigger-store**

Executing `ByNeed`

- Semantic statement is
  ({ByNeed ⟨x⟩ ⟨y⟩} , *E*)
  - ⟨x⟩ is mapped to one-argument prcedure
  - ⟨y⟩ mapped to a variable
- If ⟨y⟩ is not determined (unbound)
  - Add the trigger `trig(E(⟨x⟩) E(⟨y⟩))` to the trigger store
- If ⟨y⟩ is determined (bound)
  - Create a new thread with initial semantic stack `[({⟨x⟩ ⟨y⟩},E)]`

Executing ByNeed II

- trig(x, y) is in the trigger-store
- A need on y is detected
  - A thread suspends because of an activation condition that requires y to be determined, or
  - y is bound (made determined) by another thread
- Create a new thread with initial Semantic Stack: [({x y},∅)]
  - {…} is an apply procedure (that takes a procedure value and a variable)

Garbage Collection of Threads

- If a thread is known to be suspended forever, it can be garbage-collected
  - suspends on variable not in use by any other thread
  - does not change semantics, just saves memory
- Approximation, only straight-forward cases
  - impossible: detect whether thread will have no effect!
  - really impossible!

Summary

- Threads are organized as multiset of semantic stacks
- Thread creation inserts new semantic stack
  - inherits environment
  - shares store
- Thread termination removes threads

Agents and Message Passing Concurrency

Client-Server Architectures

- Server provides some service
  - receives message
  - replies to message
  - example: web server, mail server, …
- Clients know address of server and use service by sending messages
- Server and client run independently

Peer-to-Peer Architectures

- Similar to Client-Server:
  - every client is also a server
  - communicate by sending messages to each other
- We call all these guys (client, server, peer)

  agent

Common Features

- Agents
  - have identity — mail address
  - receive messages — mailbox
  - process messages — ordered mailbox
  - reply to messages — pre-addressed return letter

- Now how to cast into programming language?

Message Sending

- Message — data structure
- Address — port
- Mailbox — stream of messages
- Reply — dataflow variable in message

Port

- Port *address*:[*S*]
  - stores stream *S* under unique address
  - stored stream changes over time
- The stream is tail of message stream
  - sending a message *M* adds message to end

Message Sending to Port

- Port *a*:[*S*]
- Send *M* to *a*
  - read stored stream *S* from address *a*
  - create new store variable *S'*
  - bind *S* to *M*|*S'* (cons)
  - update stored stream to *S'*

Port Procedures

- Port creation

```
P={NewPort Xs}
```

- Message sending

```
{Send P X}
```


Example

```
declare S P
P={NewPort S}
{Browse S}
```

- Displays initially `S` (or _)

Example

```
declare S P
P={NewPort S}
{Browse S}
```

- Execute {Send P a}
- Shows a|_

Example

```
declare S P
P={NewPort S}
{Browse S}
```

- Execute {Send P b}
- Shows a|b|_

Question

```
declare S P
P={NewPort S}
{Browse S}
thread {Send P a} end
thread {Send P b} end
```

- What will the Browser show?

Question

```
declare S P
P={NewPort S}
{Browse S}
thread {Send P a} end
thread {Send P b} end
```

- What will the Browser show?
- Either a|b|_ or b|a|_
  - non-determinism: we can't say what

Answering Messages

- Do not reply by address, use something like pre-addressed reply envelope
  - dataflow variable!!!
- `{Send P pair(Message Answer)}`
- Receiver can bind Answer!

A Math Agent

```
proc {Math M}
   case M
   of add(N M A) then A=N+M
   [] mul(N M A) then A=N*M
   [] int(Formula A) then
      A = …
   end
end
```

Making the Agent Work

```
MP = {NewPort S}
proc {MathProcess Ms}
   case Ms of M|Mr then
      {Math M} {MathProcess Mr}
   end
end
thread {MathProcess S} end
```


Smells of Higher-Order…

```
proc {ForAll Xs P}
   case Xs
   of nil   then skip
   [] X|Xr then {P X} {ForAll Xr
  P}
   end
end
```

- Call procedure P for all elements in Xs

Smells of Higher-Order…

- Using `ForAll`, we have

```
proc {MathProcess Ms}
   {ForAll Xs Math}
end
```


Making the Agent Work

```
MP = {NewPort S}
thread {ForAll S Math} end
```

Making the Agent Work

```
MP = {NewPort S}
thread for M in S do {Math M} end
end
```

2003-10-17 S. Haridi, CS2104, L10 (slides: C. Schulte, S. Haridi) 99

Smells Even Stronger…

```
fun {NewAgent Process}
   Port Stream
in
   Port={NewPort Stream}
   thread {ForAll Process} end
   Port
end
```

2003-10-17 S. Haridi, CS2104, L10 (slides: C. Schulte, S. Haridi) 100

Why Do Agents/Processes Matter?

- Model to capture communicating entities
- Each agent is simply defined in terms of how it replies to messages
- Each agent has a thread of its own
  - no screw-up with concurrency
  - we can easily extend the model so that each agent have a state (encapsulated)
- ***Extremely useful to model systems!***

Summary

- Ports for message sending
  - use stream (list of messages) as mailbox
  - port serves as unique address
- Use agent abstraction
  - combines port with thread running agent
  - simple concurrency scheme
- Introduces non-determinism… and state!

Next Lecture

- Invited lecture
- After that: ports and agents revisited

See You In Two Weeks!

Have a Nice Weekend