LINFO1131
Concurrent programming concepts
Lecture 5

Limitations of declarative programming

Oct. 13, 2021

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Introduction

- Is declarative programming all we need?
  - We saw that declarative programming is both expressive and efficient
    - We have used higher-order programming and concurrency, and we have defined efficient ephemeral and persistent algorithms
  - Given this, do we need any nondeclarative programming concepts?
- Limitations of declarative programming
  - The answer is yes, we need nondeclarative programming concepts
  - There are some things a declarative program simply cannot do
- Interaction with the real world
  - A declarative execution is a reduction of a lambda expression
  - During this reduction there can be no interaction with the real world!
  - Why not? Let us look closely to understand what is going on…

2

Beyond declarative programming?

- Up to now we have seen three declarative paradigms
  - Sequential functional programming
  - Deterministic dataflow concurrency ("Concurrency for Dummies")
  - Lazy deterministic dataflow
- Ideally, your program should be completely declarative!
  - This gives all the advantages of declarative programming, such as simple reasoning, testing, and maintenance!
  - But unfortunately this is impossible
  - Why is it impossible? Let us see by looking at lambda calculus!

3

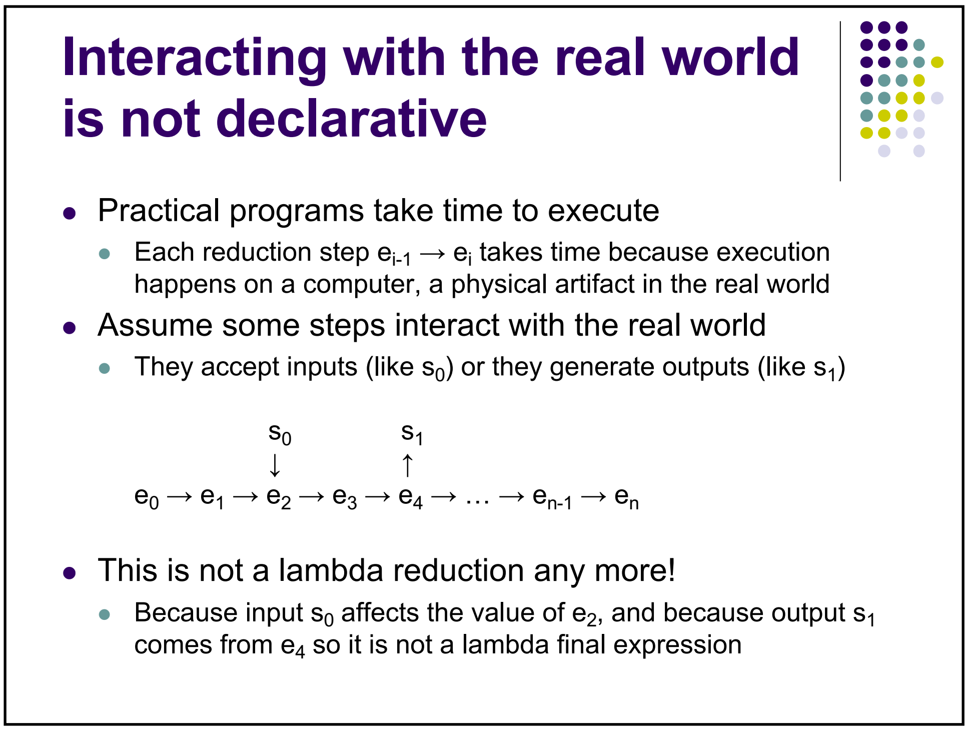
Declarative execution = lambda reduction

- Declarative execution is equivalent to lambda reduction:

$$e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ \ldots \rightarrow e_{n-1} \rightarrow e_n$$

- Execution starts with initial expression e_0 and reduces it in steps, ending with final expression e_n
  - Lambda calculus is Turing complete, it is generally expressive
  - Furthermore, it obeys the Church-Rosser theorem (confluence): the final result e_n is independent of the reduction order
- But how can this reduction interact with the real world?
  - It cannot! All information is already in the initial expression e_0, nothing is added later.

4



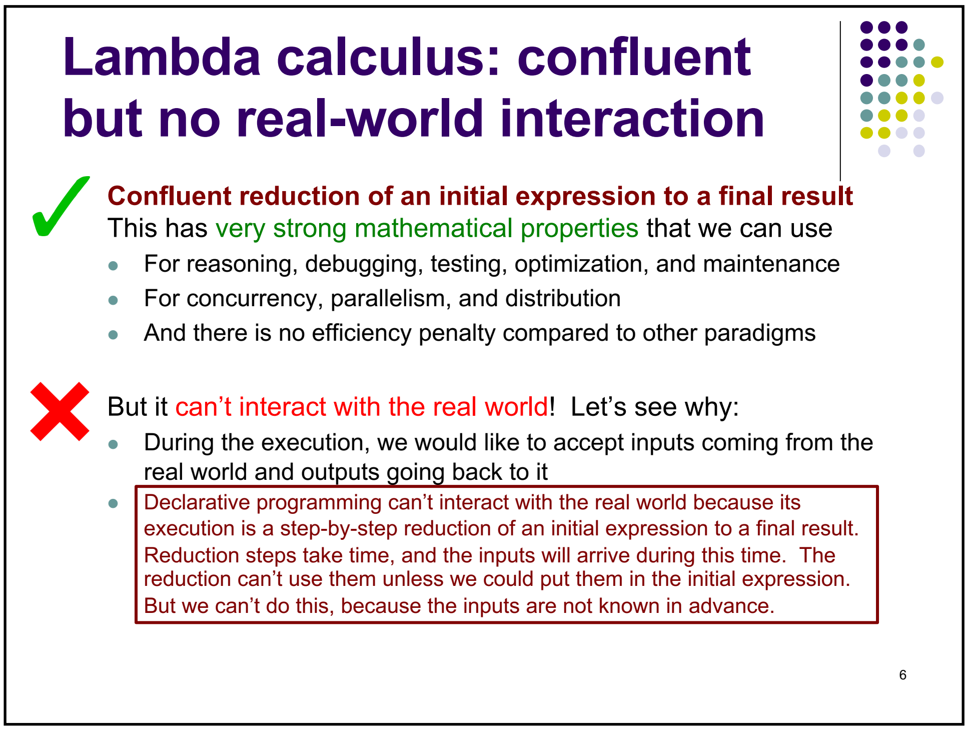
Interacting with the real world is not declarative

- Practical programs take time to execute
  - Each reduction step $e_{i-1} \rightarrow e_i$ takes time because execution happens on a computer, a physical artifact in the real world
- Assume some steps interact with the real world
  - They accept inputs (like s_0) or they generate outputs (like s_1)

$$\begin{tikzcd}
 & & s_0 \arrow[d] & & s_1 & & & \\
e_0 \arrow[r] & e_1 \arrow[r] & e_2 \arrow[r] & e_3 \arrow[r] & e_4 \arrow[r] \arrow[u] & \dots \arrow[r] & e_{n-1} \arrow[r] & e_n
\end{tikzcd}$$

- This is not a lambda reduction any more!
  - Because input s_0 affects the value of e_2, and because output s_1 comes from e_4 so it is not a lambda final expression

5



Lambda calculus: confluent but no real-world interaction

✓ **Confluent reduction of an initial expression to a final result**
This has very strong mathematical properties that we can use

- For reasoning, debugging, testing, optimization, and maintenance
- For concurrency, parallelism, and distribution
- And there is no efficiency penalty compared to other paradigms

✗ But it can't interact with the real world! Let's see why:

- During the execution, we would like to accept inputs coming from the real world and outputs going back to it
- Declarative programming can't interact with the real world because its execution is a step-by-step reduction of an initial expression to a final result. Reduction steps take time, and the inputs will arrive during this time. The reduction can't use them unless we could put them in the initial expression. But we can't do this, because the inputs are not known in advance.

6

6

Interacting with the real world

7

Imperative programming

- To interact with the real world, we need to add something to the declarative paradigms
  - A way to receive inputs and send outputs during execution
  - This is usually called imperative programming
- This lets us interact with the real world, but we also have to give up the goodness of declarative programming
- Can we have our cake and eat it too? Can we have both the good properties of declarative programming and interaction with the real world?
  - No we can't! So what can we do…?

8

The right way to design programs

- Write most of the program in a declarative paradigm
  - Then add small pieces of imperative programming only in those places that interact with the real world
  - Usually there are only a very few such places, so we keep most of the advantages of declarative programming
- We can use this to improve legacy systems too…
  - Legacy systems are often not designed like this! They do too much imperative programming. Older languages such as Java are especially bad since they encourage this.
  - This gives us a measure to judge how well legacy systems are designed (and a way to improve them: make them more declarative)

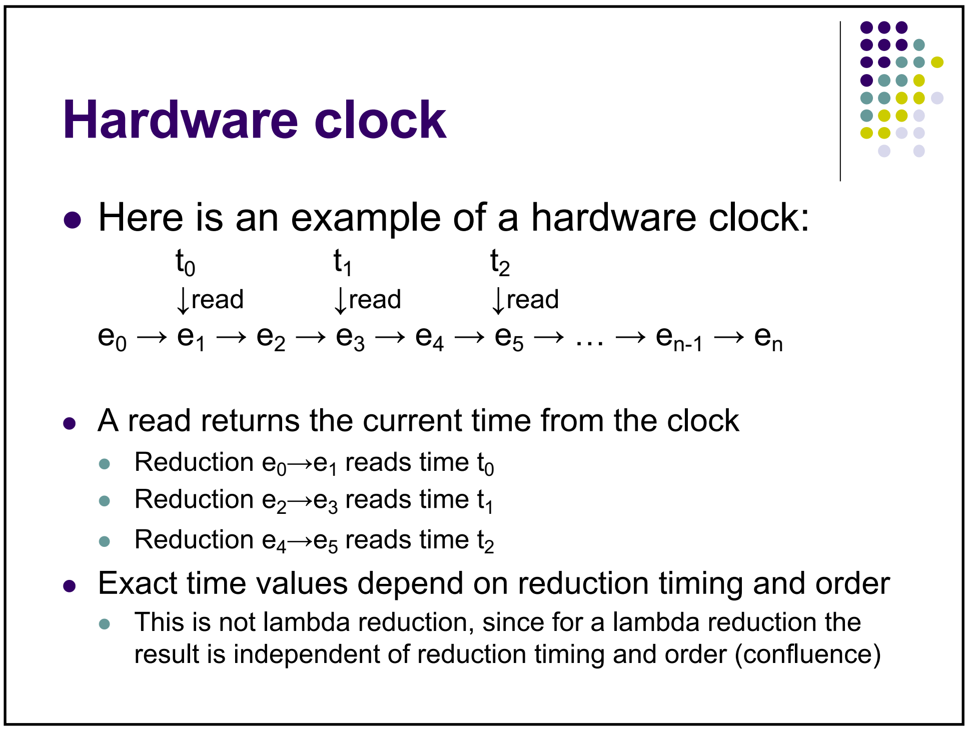
9

9

Kinds of interactions

- There are many ways that a program can interact with the real world
- Here are three typical possibilities:
  - Hardware clock: input s_a gives the clock time
  - Mutable variable: output s_a writes to a register, later input s_b (with b>a) reads from the register
  - Communication channel: output s_a sends to a channel, later input s_b receives from the channel
- Executions give different results depending on the exact timing and order of the reductions

10



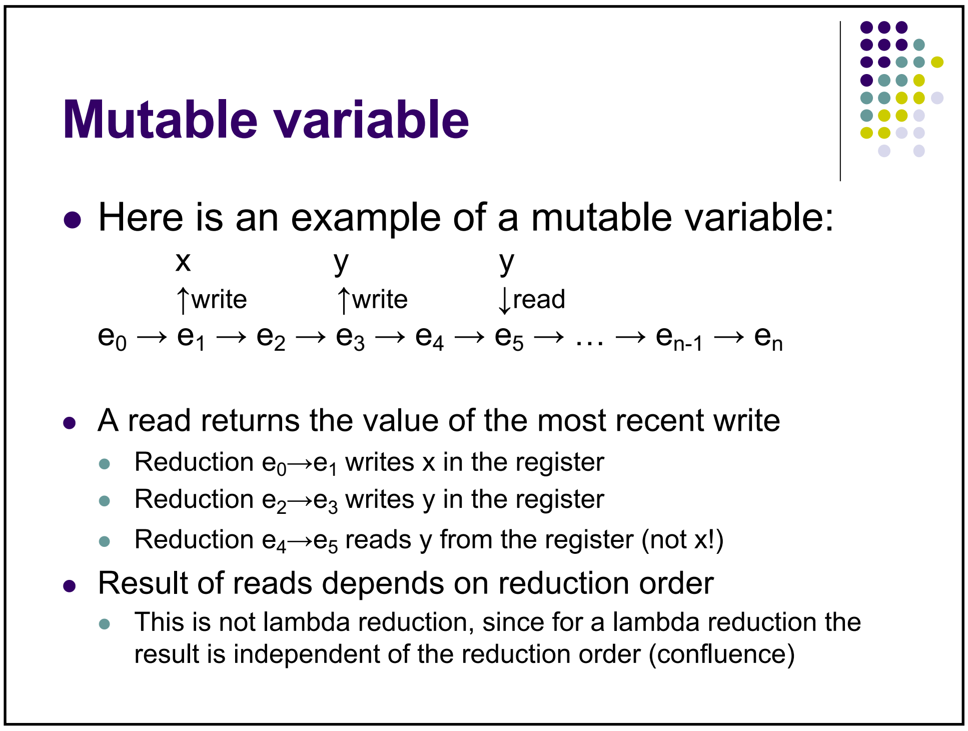
Hardware clock

- Here is an example of a hardware clock:

$$\begin{array}{cccccc} & t_0 & & t_1 & & t_2 \\ & \downarrow\text{read} & & \downarrow\text{read} & & \downarrow\text{read} \end{array}$$

$$e_0 \to e_1 \to e_2 \to e_3 \to e_4 \to e_5 \to \ldots \to e_{n-1} \to e_n$$

- A read returns the current time from the clock
  - Reduction $e_0 \to e_1$ reads time t_0
  - Reduction $e_2 \to e_3$ reads time t_1
  - Reduction $e_4 \to e_5$ reads time t_2
- Exact time values depend on reduction timing and order
  - This is not lambda reduction, since for a lambda reduction the result is independent of reduction timing and order (confluence)



Mutable variable

- Here is an example of a mutable variable:

$$\begin{array}{cccccc} & x & & y & & y \\ & \uparrow\text{write} & & \uparrow\text{write} & & \downarrow\text{read} \end{array}$$

$$e_0 \to e_1 \to e_2 \to e_3 \to e_4 \to e_5 \to \ldots \to e_{n-1} \to e_n$$

- A read returns the value of the most recent write
  - Reduction $e_0 \to e_1$ writes x in the register
  - Reduction $e_2 \to e_3$ writes y in the register
  - Reduction $e_4 \to e_5$ reads y from the register (not x!)
- Result of reads depends on reduction order
  - This is not lambda reduction, since for a lambda reduction the result is independent of the reduction order (confluence)

Communication channel

- Here is an example of a FIFO channel:

$$e_0 \rightarrow \overset{\overset{x}{\uparrow\text{send}}}{e_1} \rightarrow e_2 \rightarrow \overset{\overset{y}{\uparrow\text{send}}}{e_3} \rightarrow e_4 \rightarrow \overset{\overset{x}{\downarrow\text{receive}}}{e_5} \rightarrow \overset{\overset{y}{\downarrow\text{receive}}}{e_6} \rightarrow \ldots \rightarrow e_{n-1} \rightarrow e_n$$

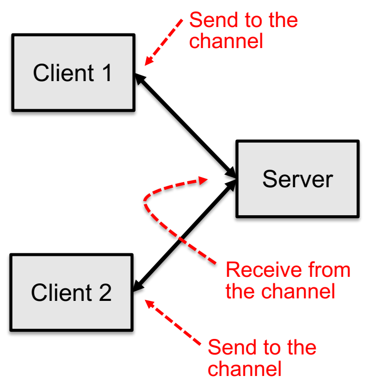
- The write happens before the read in the reduction order
  - Reduction $e_0 \rightarrow e_1$ sends x on the channel
  - Reduction $e_2 \rightarrow e_3$ sends y on the channel
  - Reduction $e_4 \rightarrow e_5$ receives x from the channel
  - Reduction $e_5 \rightarrow e_6$ receives y from the channel

Nondeterministic!

- Order of received values depends on reduction order
  - Sending of x and y might be reversed if they are concurrent!

13

Client/server application



- Let's use the communication channel to build a client/server application
  - To satisfy client liveness, the server must accept each incoming client request in a reasonable time that depends only on the travel time from the client to the server
- However, the order of the requests cannot be determined in advance because it depends on precise client timing (different timings give different reduction orders)
  - The communication channel is nondeterministic (not declarative)
- The whole client/server application is therefore nondeclarative, even if all the other code is purely declarative

14

14

The two most important nondeclarative operations

- Two important nondeclarative operations are mutable variables and communication channels
  - In the course we will show how to use both of them
- Mutable variables: we call them cells
  - Leads to shared-state concurrency (as in Java)
  - Typical concepts: locks, monitors, transactions
- Communication channels: we call them ports
  - Leads to message-passing concurrency (as in Erlang)
  - Techniques of multi-agent programming

15

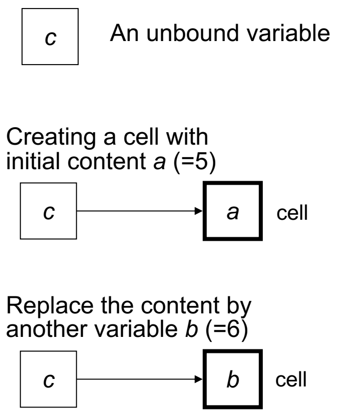
Cells

16

A cell

- To overcome the limitations of declarative programming, we add cells (mutable variables) to the language
- A cell is a box with identity and content
  - The identity is a constant (the “name” or “address” of the cell)
  - The content is a variable (in the single-assignment store)
- The content can be replaced by another variable

```
A=5
B=6
C={NewCell A}  % Create a cell
{Browse @C}    % Display content
C:=B           % Change content
{Browse @C}    % Display content
```



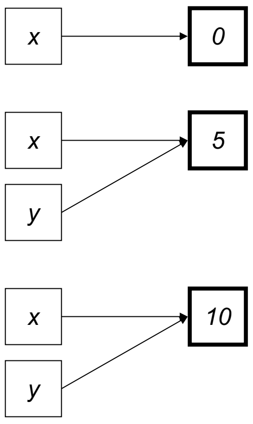
c
An unbound variable
Creating a cell with initial content *a* (=5)
c
a
cell
Replace the content by another variable *b* (=6)
c
b
cell

Adding cells to the kernel language

- We add cells and their operations
  - Cells have three operations
- C={NewCell A}
  - Create a new cell with initial content A
  - Bind C to the cell’s identity
- C:=B
  - Check that C is bound to a cell’s identity
  - Replace the cell’s content by B
- Z=@C
  - Check that C is bound to a cell’s identity
  - Bind Z to the cell’s content

Cell examples (1)

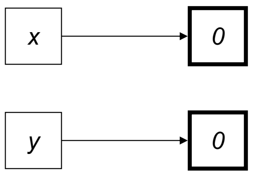
- X={NewCell 0}
- X:=5
- Y=X
- Y:=10
- @X==10 % true
- X==Y % true



19

Cell examples (2)

- X={NewCell 0}
- Y={NewCell 0}
- X==Y % false
- Because X and Y refer to different cells, with different identities
- @X==@Y % true
- Because the contents of X and Y are the same value



20

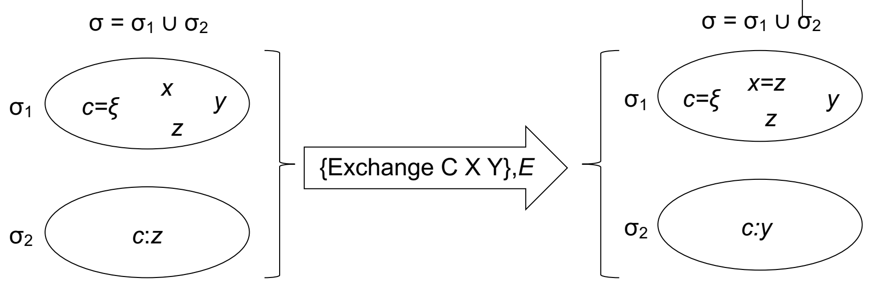
Cell semantics (1)

- Assume single-assignment store σ_1 with variables
- Assume cell store σ_2 that contains pairs of variables
- Full store $\sigma = \sigma_1 \cup \sigma_2$

environment

- C={NewCell X}, {C→*c*, X→*x*}
  - Assume variables *c*, *x* ∈ σ_1 (*c* is unbound)
  - Create fresh name ξ, bind *c*=ξ, add pair *c*:*x* to σ_2
- {Exchange C X Y}, {C→*c*, X→*x*, Y→*y*}
  - Assume *c*=ξ, variables *x*, *y*, *z* ∈ σ_1 , *c*:*z* ∈ σ_2
  - Bind *x*=*z* (get old value), update pair to *c*:*y* (update new value)
  - Combines the two operations X=@C and C:=Y

Cell semantics (2)



- {Exchange C X Y} binds x to the cell's old content z and updates the new cell content to y
  - The exchange operation is atomic, which means the scheduler is guaranteed never to stop in the middle, it happens as one indivisible step
- We assume that environment *E*={C→*c*,X→*x*,Y→*y*}

Ports

23

A port (named stream)

- To overcome the limitations of declarative programming, we add ports (named streams) to the language
- Ports have two operations:

```
P={NewPort S}    % Create port P with stream S
{Send P X}       % Add X to end of port P's stream
```

- How does this solve the client/server program?
  - With a million clients C_1 to $C_{1000000}$:
    Each client C_i does {Send P M_i} for each message it sends
  - The server reads the stream S, which contains all messages from all clients in some nondeterministic order

24

Port examples

- We create a port and do sends:

```
P={NewPort S}
{Browse S} % Displays _
{Send P a}  % Displays a|_
{Send P b}  % Displays a|b|_
```

- What happens if we do:

```
thread {Send P c} end
thread {Send P d} end
```

- What are the possible results of these two sends for all choices of the scheduler?

25

Port semantics (1)

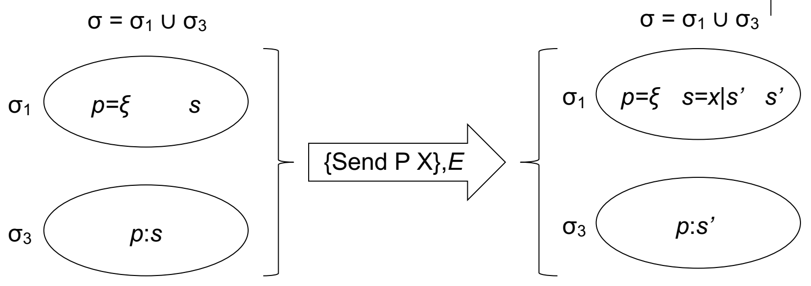
- Assume single-assignment store σ_1 with variables
- Assume a port store σ_3 that contains pairs of variables
  - (Remember σ_2 is the cell store we introduced before)

environment

- P={NewPort S}, $\{P \rightarrow p, S \rightarrow s\}$
  - Assume unbound variables p, $s \in \sigma_1$
  - Create fresh name ξ, bind $p=\xi$, add pair $p{:}s$ to σ_3
- {Send P X}, $\{P \rightarrow p, X \rightarrow x\}$
  - Assume $p=\xi$, unbound variable $s \in \sigma_1$, $p{:}s \in \sigma_3$
  - Create fresh unbound variable s', bind $s=x|s'$, update pair to $p{:}s'$

26

Port semantics (2)



- {Send P X} adds x to the end of the port's stream and updates the new end of stream
  - The send operation is atomic, which means the scheduler is guaranteed never to stop in the middle, so it happens as if it is one indivisible step
- We assume that environment E={P→p,X→x}

27

Cell + port semantics summary

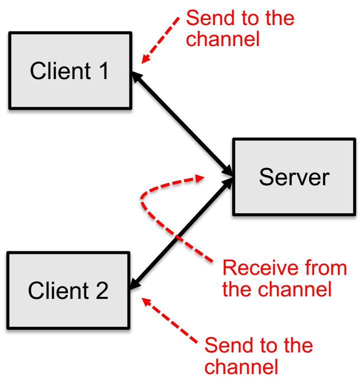
- The full store $\sigma = \sigma_1 \cup (\sigma_2 \cup \sigma_3)$ has two parts:
  - Single-assignment store (contains variables)
    σ_1 = {t, u, v, x=ξ, y=ζ, z=10, w=5}
  - Multiple-assignment store (contains pairs)
    $(\sigma_2 \cup \sigma_3)$ = {x:t, y:w}
- The multiple assignment store has two kinds of nondeclarative entities
  - Cells: mutable variables
  - Ports: communication channels

28

Client/server application redux

29

Client/server application redux



- We can use the communication channel to build a client/server application
  - To satisfy client liveness, the server must accept each incoming client request in a reasonable time that depends only on the travel time from the client to the server
- We can satisfy this requirement by using one communication channel
  - The communication channel is nondeterministic (not declarative)
- Only one communication channel is needed for the whole client/server; all other code can be declarative

30

30

Client/server with a port

- We implement client/server communication with a port
  P={NewPort S} % Create a new port P with stream S
- Client code: (any number of clients!)
  {Send P M} % Send message M to port P
- Server code:

```
 proc {Server S}
      case S of M|T then
            … % Handle message M
            {Server T}
      end
end
{Server S} % Run server with stream S
```

31

Conclusions

32

Conclusions

- Declarative paradigms are powerful but cannot always be used
  - Declarative paradigms are based on lambda calculus, which makes them confluent but also means they cannot interact with the real world
- To interact with the real world, we extend declarative paradigms with imperative language concepts such as mutable variables or communication channels
  - Mutable variables (cells) lead to shared-state concurrency (Java)
  - Communication channels (ports) lead to message-passing concurrency (Erlang)
- To keep the advantages of declarative programming as much as possible, programs should be mostly declarative and only add imperative entities to interact with the real world outside the application
  - In most cases, only a very small number of imperative entities are needed.

33