# Concurrent programming with dataflow variables

Oleg Batrashev
Distributed Systems Group
University of Tartu

January 31, 2009

# Outline

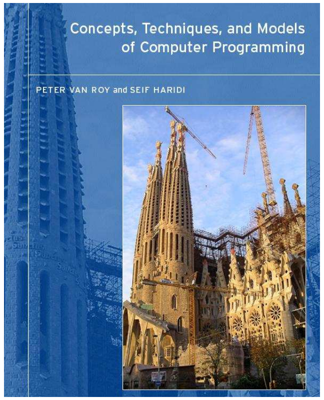


- Oz language
  - syntax of declarative model
  - dataflow variables
  - addition: threads, ports, cells
- Programming with Oz
  - in declarative model
  - in multiagent dataflow model
  - in distributed programming
- Example: P2P chat
- Conclusions

# Motivation



- I wanted to study concurrent programming
  - was suggested the book "*Concepts, Techniques, and Models of Computer Programming*" by Peter van Roy and Seif Haridi




- give a course on concurrent programming languages
  - give a quick overview of some topics
  - suggestions, remarks are welcome

# Justification



of non-theoretical talk on Oz

- the paper "*A concurrent lambda calculus with futures.*" by J. Hiehren, J.Schwinghammer, G. Smolka, 2006

> Many ideas in Alice ML (except those for typing) are inspired by, and inherited from, the concurrent constraint programming language Mozart-Oz.

- future is a read-only dataflow variable

# Oz language

# Oz (declarative) kernel



- Statements
  - `skip`
  - $\langle s_1 \rangle \langle s_2 \rangle$
  - `local` $\langle x \rangle$ `in` $\langle s \rangle$ `end`
  - $\langle x_1 \rangle$ `=` $\langle x_2 \rangle$
  - $\langle x \rangle$ `=` $\langle v \rangle$
  - `if` $\langle x \rangle$ `then` $\langle s_1 \rangle$ `else` $\langle s_2 \rangle$ `end`
  - `case` $\langle x \rangle$ `of` $\langle ptn \rangle$ `then` $\langle s_1 \rangle$ `else` $\langle s_2 \rangle$ `end`
  - `proc {`$\langle x \rangle$ $\langle y_1 \rangle$ `...` $\langle y_n \rangle$`} end`
  - `{`$\langle x \rangle$ $\langle y_1 \rangle$ `...` $\langle y_n \rangle$`}`
- Atom (symbolic constant)

  `person nil true false 'with spaces' '|'`
- Record (label with a set of feature/value pairs)

  $\langle label \rangle (1 : \langle x_1 \rangle \ldots n : \langle x_n \rangle \; a_1 : \langle x_{n+1} \rangle \ldots a_m : \langle x_{n+m} \rangle$

  `person(1:"Oleg" 2:male city:Tartu year:2009)`

# Dataflow variables



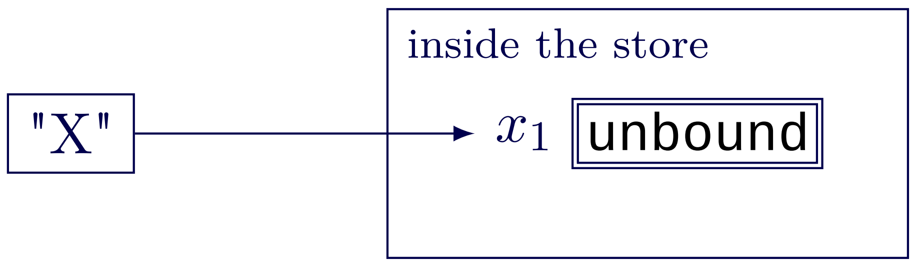
- single-assignment
  `X=5 X=6` throws exception
- logical, i.e. may be
  - unbound
    ```
    declare X
    local X in ...  end
    ```
  - bound to a value
    `X=5`
  - bound to another variable
    `X=Y`
- `=` is unification, not assignment
  `X=5` is equivalent to `5=X`
- unification goes recursively both ways
  ```
  person(name:X age:15) = L(name:"George" age:Y)
  ```

# Single-assignment store

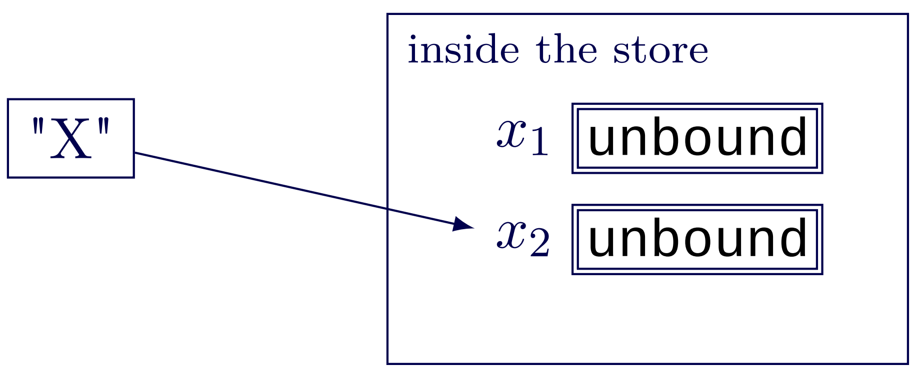


Conceptually (implementation may be more optimal),

- `declare X` maps *variable identifier* $X$ to the new *variable* $x_1$ in the store



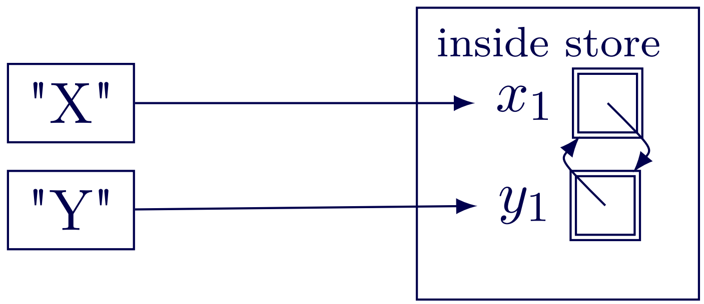

- the following `declare X` maps $X$ to the new variable $x_2$



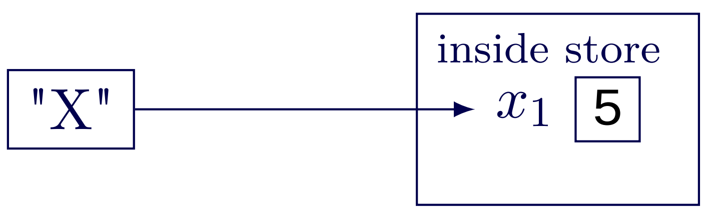
# Variables and values



- bound variables `X=Y` become indistinguishable
  - change of $X$ ($x_1$) relfects on $Y$ ($y_1$) and v.v.

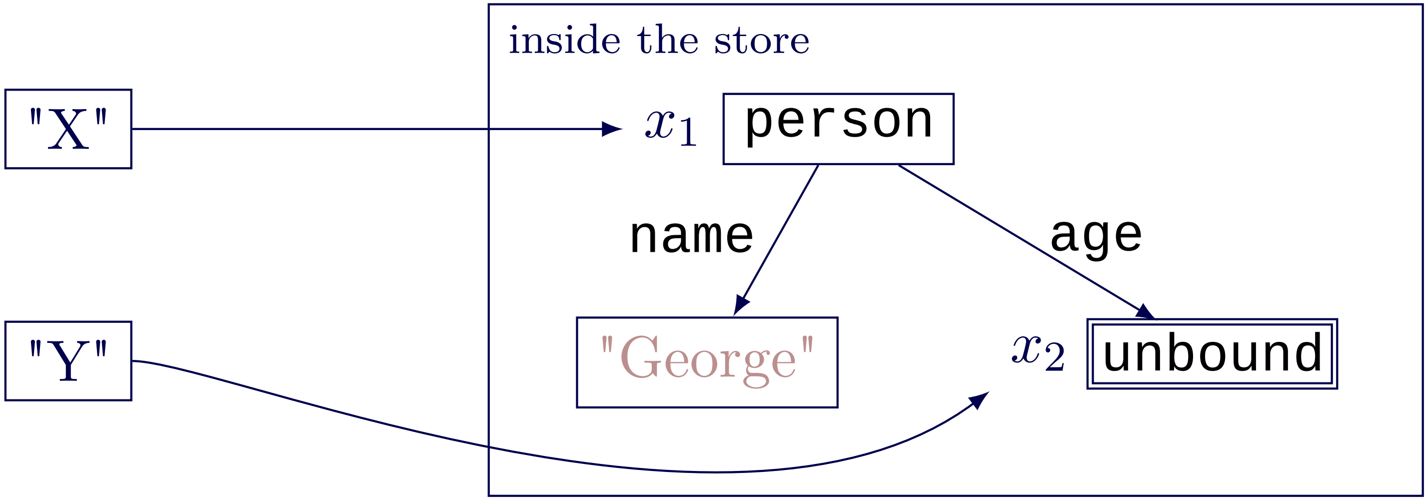



- variable bound to a value is just the value
  - $x_1$ becomes “unneeded”



```
declare X Y
X=person(name:"George" age:Y)
```

results in




- binding `Y=6` results in `X` equal to `person(name:"George" age:6)`

- Reading unbound variable value *blocks* until it is bound
- Some operations require value
  - operators +,-,*, etc.
    ```
    declare X Y
    Y=X+1
    ```
  - condition in *if* statement
  - value and pattern in *case* statement
- Many operations do not require value
  - save variable into data structure
    ```
    declare X Y=person(name:_)
    Y.name=X
    X="Richard"
    ```
  - send variable over a network

# Ports



- in Oz *streams* are lists with unbound tail
  `Ls=1|2|5|Xs`
- to extend we bind the tail to cons record
  `Xs='|'(7 Xs2)` (with operator `Xs=7|Xs2`)
  - where *Xs2* is a new tail
- The problem
  - several threads can read the tail
    `case Xs of X|Xs2 then`
  - non-deterministic append is not possible
- Port - abstraction with a stream
  - `{NewPort Ls ?Port}` - returns port
  - `{Send Port X}` - non-det binds the tail

- exceptions
  - `raise` $\langle x \rangle$ `end`
  - `try` $\langle s_1 \rangle$ `catch` $\langle s_2 \rangle$ `end`
- cells (mutable variables)
  - `{NewCell X ?C}`
  - `X=@C` - get value
  - `C:=X` - set value
  - `X=C:=Y` - exchange value

# Programming in Oz

# Declarative model



```
fun {AppendF Xs Ys}
   case Xs
   of nil then Ys
   [] X|Xs2 then X|{AppendF Xs2 Ys}
   end
end
proc {AppendP Xs Ys Zs}
   case Xs
   of nil then Zs=Ys
   [] X|Xs2 then Zs2 in
      {AppendP Xs2 Ys Zs2}
      Zs=X|Zs2
   end
end
```

- switch last 2 lines to create a hole

# Comparison with functional



- Declarative programming in Oz
  - much like functional
  - but procedural
  - slightly less restrictive
    - can create holes and fill in other places
  - slightly more error-prone
    - may forget to fill a hole
  - still completely deterministic

```
proc {PMap F Xs ?Zs}
   case Xs
   of nil then Zs=nil
   [] X|Xs2 then Zs2 in
      Zs=thread {F X} end|Zs2
      thread {PMap F Xs2 Zs2} end
   end
end
```

- Can insert thread construct at any place
  - nothing ever breaks
  - unless exceptions are there too

# Port objects (Agents)



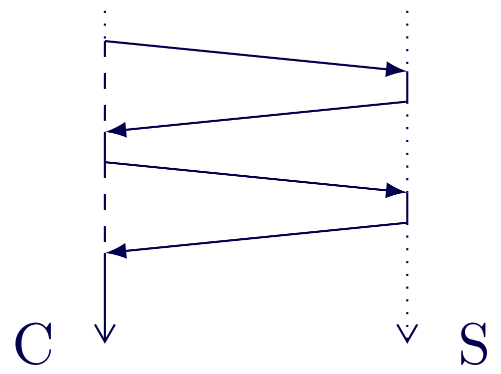
- A *single* thread
  - reads messages from a stream (list)
  - after a message is processed changes state
- Can be done with infinite recursion on a list
- Shorter to define with *FoldL*
  - *Sin* - input stream
  - *Fun* - transform function

```
fun {NewPortObject Init Fun}
   Sin Sout in
   thread {FoldL Sin Fun Init Sout} end
   {NewPort Sin}
end
```

# RMI (1)



- synchronous




```
proc {ServerProc Msg}
   case Msg
   of calc(X Y) then
      Y=X*X+5.0*X+6.0
   end
end
Server={NewPortObject2 ServerProc}
```

# RMI (2)

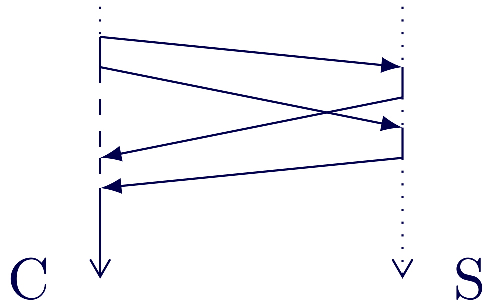


```
proc {ClientProc Msg}
   case Msg
   of work(Y) then Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Wait Y1}
      {Send Server calc(20.0 Y2)}
      {Wait Y2}
      Y=Y1+Y2
   end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

- *Wait* returns when the argument is bound to a value

# Asynchronous RMI

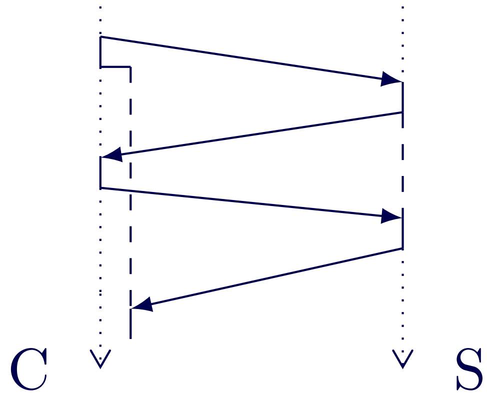





```
proc {AClientProc Msg}
   case Msg
   of work(Y) then Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Send Server calc(20.0 Y2)}
      Y=Y1+Y2
   end
end
Client={NewPortObject2 AClientProc}
{Browse {Send Client work($)}}
```

# RMI with callback u/ thread (1)






```
proc {ServerProc Msg}
   case Msg
   of calc(X ?Y Client) then X1 D in
      {Send Client delta(D)}
      X1=X+D
      Y=X1*X1+2.0*X1+2.0
   end
end
Server={NewPortObject2 ServerProc}
```

# RMI with callback u/ thread (2)



```
proc {ClientProc Msg}
   case Msg
   of work(?Z) then Y in
      {Send Server calc(10.0 Y Client)}
      thread Z=Y+100.0 end
   [] delta(?D) then
      D=1.0
   end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

- adding new protocols is easy
- and fun

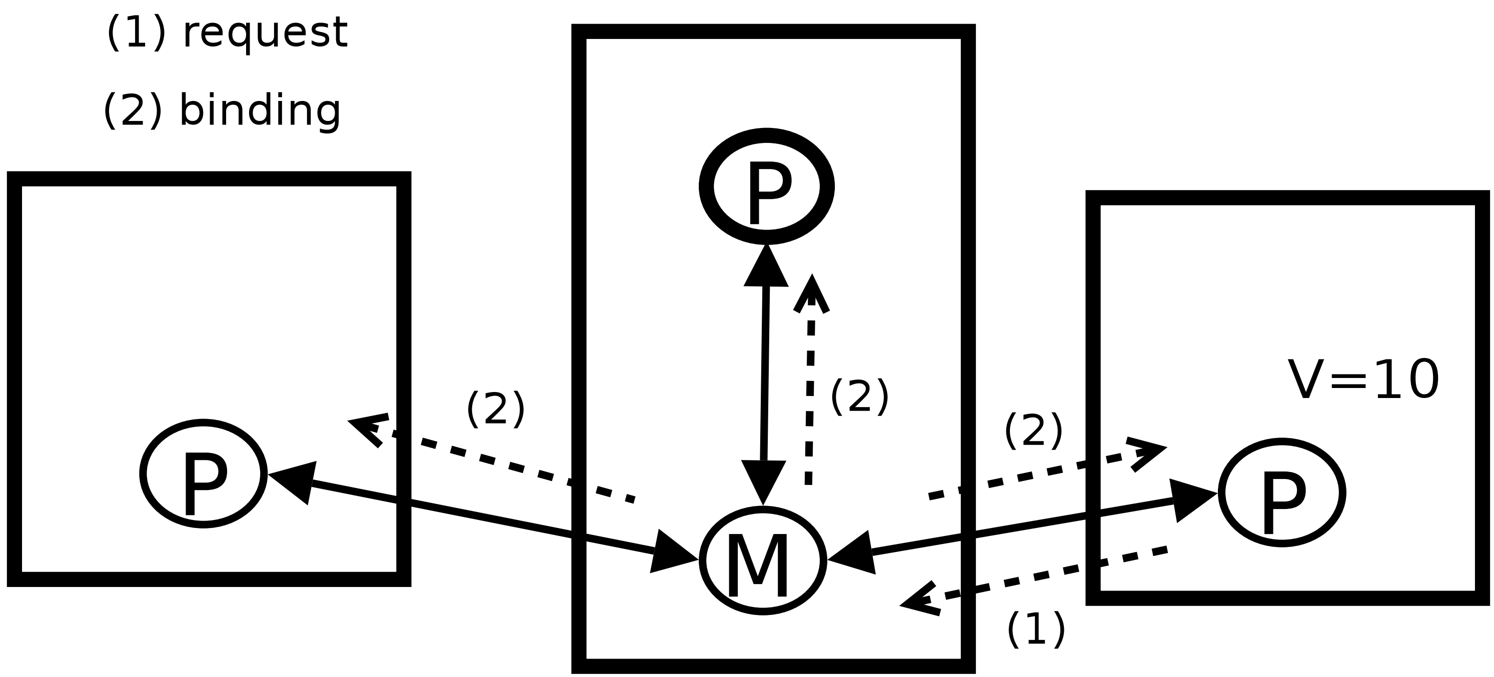
Mozart is an Oz VM + libraries

- network transparency
  - almost no changes of code to distribute
- network awareness
  - can change entity (DV, cell, port, value) distribution protocols
  - protocols: stationary, mobile, eager/lazy copying, ...
- openness
- fault tolerance
  - can install asyncronous watchers on entities

# Protocol for DV



- Manager on one site
- Proxies on all sites



# Example: P2P chat

# Multiport object



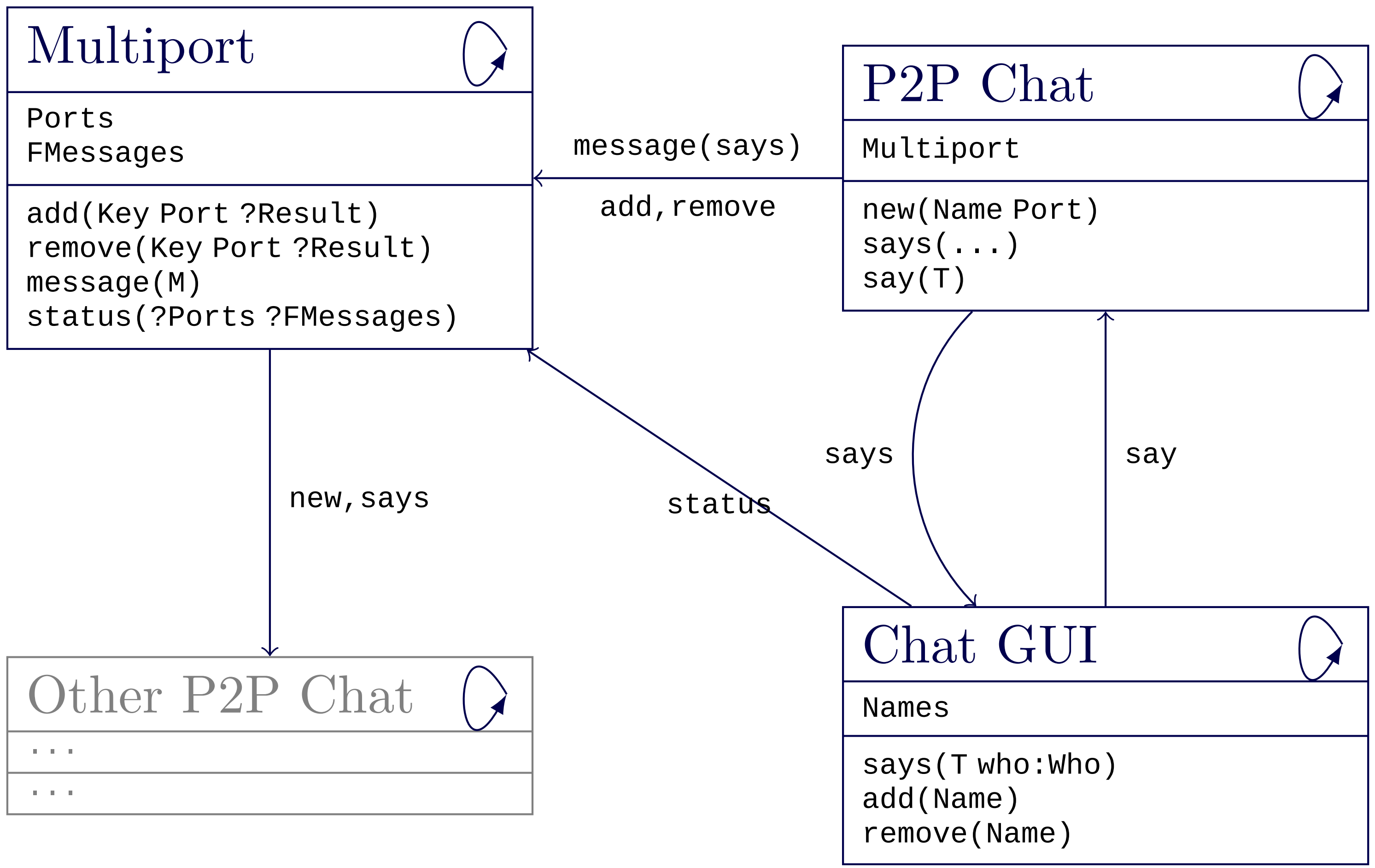
- port object (agent) with state
- maintains list of ports
- broadcasts messages

| Multiport |
| --- |
| `Ports`<br>`FMessages` |
| `add(Key Port ?Result)`<br>`remove(Key Port ?Result)`<br>`message(M)`<br>`status(?Ports ?FMessages)` |

- on `add(Key Port ?Result)`
  - adds *Port* with name *Key* to the list
  - binds *Result* to true or false

# Agents




Multiport
Ports
FMessages
add(Key Port ?Result)
remove(Key Port ?Result)
message(M)
status(?Ports ?FMessages)
P2P Chat
Multiport
new(Name Port)
says(...)
say(T)
message(says)
add,remove
new,says
status
says
say
Other P2P Chat
...
...
Chat GUI
Names
says(T who:Who)
add(Name)
remove(Name)

# Track connected users

- GUI needs to show connected peers
  - constantly update it
- Multiport holds unbound variable for future events (messages)
  - shares on `status(?Ports ?FMessages)` message

```
fun {MultiPortProc State Message}
   state(ports:Ps futureMessages:Fs) = State
   NewFs
in
   Message|NewFs = Fs
   ...
   state(ports:Key#P|Ps futureMessages:NewFs)
end
```

# Conclusions

# Wrap up



- concurrent programming can be simple and fun
  - without mutable variables
- Oz/Mozart is a great platform to play with concurrent and distributed programming

Limitations

- futures vs logic variables
  - allow to determine dataflow
  - static dataflow -> static type inference
- dataflow variables are not good for theory
  - are futures as expressive?