# Type Systems

Vasco T. Vasconcelos

GLOBAN 2006
THE GLOBAL COMPUTING APPROACH TO ANALYSIS OF SYSTEMS
International Summer School at DTU, August 21-25, 2006

# Process calculi - what for?

- You should now this by now...
- We are interested in process calculi as core languages where to study the phenomena of concurrency

# Type systems - what for?

the focus of this lecture

- Early identification of potential runtime errors
- Imposition of a programming discipline
- Partial specification of applications

the focus of this lecture

- Uncovering important information for compilers

# Outline

- A pi-calculus
- Simply typed pi-calculus
- Input-output types
- Linear types
- Session types

# A pi-calculus

# Syntax

- Lowercase letters denote channels (or names)
- Uppercase letters represent processes
- Syntax of processes:

write on channel $x$ — $x$ is local to $P$ — parallel composition

$$P \;::=\; x![y].P \;\mid\; x?(y).P \;\mid\; (\nu x)P \;\mid\; x?*(y).P \;\mid\; P \mid P \;\mid\; \mathbf{0}$$

read on channel $x$ — inaction

# Reduction

- Communicating name $y$ on channel $x$

reduces to; evolves to

$$x![y].\mathbf{0} \mid x?(z).z![v].\mathbf{0} \to \mathbf{0} \mid y![v].\mathbf{0}$$

Rule for interaction

$$\frac{}{x![y].P \mid x?(z).Q \to P[y/z]}$$

replace $y$ by $z$ in $P$

- Communication in the presence of process $R$

$$x![y].\mathbf{0} \mid x?(z).z![v].\mathbf{0} \mid R \;\to\; \mathbf{0} \mid y![v].\mathbf{0} \mid R$$

Rule for parallel composition

$$\frac{P \to Q}{P \mid R \to Q \mid R}$$

# Reduction - more rules

- Communication in the presence of restriction

$$(\nu y)(x![y].\mathbf{0} \mid x?(z).z![v].\mathbf{0}) \;\rightarrow\; (\nu y)(\mathbf{0} \mid y![v].\mathbf{0})$$

Rule for name restriction

$$\frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q}$$

- What about this case?

$$(\nu y)(\underline{x![y]}.\mathbf{0}) \mid \underline{x?(z)}.z![v].\mathbf{0} \;\rightarrow\; ???$$

$y$ not free in this process

- We say

$$(\nu y)(x![y].\mathbf{0}\underline{)} \mid x?(z).z![v].\mathbf{0} \;\equiv\; (\nu y)(x![y].\mathbf{0} \mid x?(z).z![v].\mathbf{0}\underline{)}$$

# Structural congruence

- The last rule, structural congruence

$$\frac{P \equiv P' \to Q' \equiv Q}{P \to Q}$$

- The structural congruence relation

$$\begin{aligned}
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
P \mid Q &\equiv Q \mid P \\
P \mid \mathbf{0} &\equiv P \\
(\nu x)\mathbf{0} &\equiv \mathbf{0} \\
(\nu x)(P \mid Q) &\equiv P \mid (\nu x)Q \qquad \text{if } x \notin \mathrm{fn}(P) \\
!P &\equiv P \mid !P
\end{aligned}$$

# Example of reduction

- Show that

$$(\nu x)(x?(y).x?(z).y![z].\mathbf{0} \mid x?(w).x?(v).v![w].\mathbf{0} \mid x![a].x![b].\mathbf{0})$$

reduces to

$$(\nu x)(a![b].\mathbf{0} \mid a![w].\mathbf{0})$$

and also to

$$(\nu x)(b![a].\mathbf{0} \mid x?(y).x?(z).y![z].\mathbf{0})$$

# What can go wrong?

- Nothing!
- If only we had primitive types (and operations on them)...

$$a![2] \mid a?(x).\text{if } x \text{ then } P \text{ else } Q$$

- Instead, we shall use a polyadic pi-calculus

$$P \;::=\; x![y_1 \ldots y_n].P \;\mid\; x?(y_1 \ldots y_n).P \;\mid\; \ldots \qquad (n \geq 0)$$

and define our own data

# Data in the polyadic pi-calculus

- Boolean values are processes that follow a simple protocol

$$\text{True}\ (b) \stackrel{\text{def}}{=} b?(t\,f).t![].\mathbf{0}$$

$$\text{False}\ (b) \stackrel{\text{def}}{=} b?(t\,f).f![].\mathbf{0}$$

I am the truth value false!

- A conditional process can be written

$$\text{if } b \text{ then } P \text{ else } Q \stackrel{\text{def}}{=} (\nu xy)(b![xy] \mid x?().P \mid y?().Q)$$

- Example:

a “dead” process

$$\text{True}\ (b) \mid \text{if } b \text{ then } P \text{ else } Q \rightarrow^2 P \mid (\nu f)(f?().Q)$$

# We now have errors

- Arity mismatch. Immediate:

$$a![uw].\mathbf{0} \mid a?(z).\mathbf{0}$$

and after reduction:

$$u![a].w?(x).x![uw].\mathbf{0} \mid w![a].\mathbf{0} \mid u?(y).y?(z).\mathbf{0}$$

- In general, a process is an error when it reduces to

$$(\nu\vec{w})(x![y_1 \dots y_n].P \mid x?(z_1 \dots z_m).Q \mid R) \qquad \text{with } n \neq m$$

- Types to the rescue!

# Simpy typed pi-calculus

# Filtering out errors

- Predicate “P is an error” is undecidable, in general.
- Aim: define a (decidable) predicate $\vdash$ such that

$$\text{if } \Gamma \vdash P, \text{ then } P \text{ is not an error}$$

# Types

- Assigned to names, not to processes
- Describe what kind of names a name carries. Syntax:

$$T \ ::= \ \sharp[\vec{T}]$$

- Example:

$$\begin{aligned} \text{True}\ (b) &\stackrel{\text{def}}{=} b?(t\,f).t![].\mathbf{0} \\ t &: \sharp[] \\ f &: \sharp[] \\ b &: \sharp[\sharp[]\sharp[]] \end{aligned}$$

# Typings

- Type environments, typings in short, associate types to names

$$\Gamma \ ::= \ x_1 : T_1, \ldots, x_n : T_n$$

  and describe the types for the free names in a process

- Sequent $\Gamma \vdash P$ reads "process $P$ is well-typed in typing $\Gamma$"

- We say that "$P$ is typable" when $\Gamma \vdash P$, for some $\Gamma$

- Example

$$b \colon \sharp[\sharp[]\sharp[]] \vdash b?(tf).t![].\mathbf{0}$$

# The rules of the typing system

- Rule for names

$$\frac{}{\Gamma, \vec{x} : \vec{T} \vdash \vec{x} : \vec{T}}$$

- Rules for processes

$$\frac{}{\Gamma \vdash \mathbf{0}}$$

$$\frac{\Gamma, x : T \vdash P}{\Gamma \vdash (\nu x)P}$$

$$\frac{\Gamma \vdash x : \sharp[\vec{T}] \qquad \Gamma \vdash \vec{y} : \vec{T} \qquad \Gamma \vdash P}{\Gamma \vdash x![\vec{y}].P}$$

$$\frac{\Gamma \vdash x : \sharp[\vec{T}] \qquad \Gamma, \vec{y} : T \vdash P}{\Gamma \vdash x?(\vec{y}).P}$$

Similarly for $x? * (\vec{y}).P$

# Types, types not

- Show that the following sequent holds

$$b\colon \sharp[\sharp[]\sharp[]] \vdash b?(t\,f).t![].\mathbf{0}$$

- But that the process below is not typable

$$u![a].w?(x).x![uw].\mathbf{0} \mid w![a].\mathbf{0} \mid u?(y).y?(z).\mathbf{0}$$

# Main result

If $P$ is typable then $P$ is not an error

$$(\Gamma \vdash P \ \wedge \ P \rightarrow^{*} (\nu\vec{w})\,(x![y_1 \ldots y_n].P \mid x?(z_1 \ldots z_m).Q \mid R)) \ \Rightarrow \ n = m$$

- Proven in two parts
  - Subject Reduction

$$(\Gamma \vdash P \ \wedge \ P \rightarrow Q) \ \Rightarrow \ \Gamma \vdash Q$$

  - Type Safety

$$\Gamma \vdash (\nu\vec{w})\,(x![y_1 \ldots y_n].P \mid x?(z_1 \ldots z_m).Q \mid R) \ \Rightarrow \ n = m$$

Input-output types

# Phishing my credit card number

```
(υ myPrinter)(
   myPrinter?(doc). Print |
   myPrinter![1456854012743869] |
   someone![myPrinter]
) |
someone?(x). x?(doc). UseMyDocs
```

- What went wrong?
- Nothing, really! Only that printer channels are supposed to be written, not read

# Distinguish input from output

- But we never said that!
- We say it now: we shall distinguish between input and output types
- New syntax for types

input-output/
read-write

$$T \; ::= \; ?[\vec{T}] \;\mid\; ![\vec{T}] \;\mid\; \sharp[\vec{T}]$$

input/
read

output/
write

# Meeting expectations

- I am expecting a read-only channel; I am given a read-write channel. Shall I accept it?
- Sure! I shall use what I need (the read capability), and forget the rest (the write capability)
- A read-write channel is a subtype of a read-only channel
- If $S$ is a subtype of $T$, then an expression of type $S$ can always replace an expression of type $T$

# Subtyping

- Subtyping is a preorder on types. If $S$ is a subtype of $T$, then a channel of type $S$ is also a channel of type $T$

- Rules

$$\frac{}{T \leq T} \qquad \frac{S \leq S' \quad S' \leq T}{S \leq T}$$

$$\frac{}{\sharp[\vec{T}] \leq ?[\vec{T}]} \qquad \frac{\vec{S} \leq \vec{T}}{?[\vec{S}] \leq ?[\vec{T}]}$$

$$\frac{}{\sharp[\vec{T}] \leq ![\vec{T}]} \qquad \frac{\vec{S} \leq \vec{T}}{![\vec{T}] \leq ![\vec{S}]}$$

$$\frac{\vec{T} \leq \vec{S} \quad \vec{S} \leq \vec{T}}{\sharp[\vec{S}] \leq \sharp[\vec{T}]}$$

# New typing rules

- Old and new typing rules for names

$$\frac{}{\Gamma, \vec{x}\colon \vec{T} \vdash \vec{x}\colon \vec{T}} \qquad \frac{\Gamma \vdash \vec{x}\colon \vec{S} \quad \vec{S} \leq \vec{T}}{\Gamma \vdash \vec{x}\colon \vec{T}}$$

- Replacement rules for input and for output

was #

$$\frac{\Gamma \vdash x\colon ?[\vec{T}] \quad \Gamma, \vec{y}\colon \vec{T} \vdash P}{\Gamma \vdash x?(\vec{y}).P}$$

was #

$$\frac{\Gamma \vdash x\colon ![\vec{T}] \quad \Gamma \vdash \vec{y}\colon \vec{T} \quad \Gamma \vdash P}{\Gamma \vdash x![\vec{y}].P}$$

# The phisher is not typable

- The phisher

someone?(x). x?(doc). UseMyDocs

- The intended types

printer channels are to be written-only

doc: PostScript

x: ![PostScript]

someone: #[![PostScript]]

- Show that

someone: #[![PostScript]] $\nvdash$

someone?(x). x?(doc). 0

# Input-output types good for

- Preventing programming mistakes (illegal accesses to credit card numbers)
- Yield more powerful techniques: more processes can be deemed equivalent if one considers contexts that follow the i/o discipline

Linear types

# A lock manager

- The manager

$$LM = aquireLock?(r).\; (\nu done)(r![done].\; done?().\; LM)$$

- A client

done![]

$$(\nu s)(aquireLock![s].\; s?(done).\; CriticalRegion)$$

- Problems when CriticalRegion
  - does not release the lock - no other process will obtain the lock
  - releases the lock twice - not really an error, but ...

# Channels that should be used exactly once

- done is a channel that should be used exactly
  - Once for reading - in the Lock Manager, and
  - Once for writing - in each client
- We need more type constructors. Syntax:

$$T \ ::= \ l_\sharp[\vec{T}] \ \mid \ l_?[\vec{T}] \ \mid \ l_![\vec{T}] \ \mid \ \sharp[\vec{T}] \ \mid \ ?[\vec{T}] \ \mid \ ![\vec{T}]$$

$l$ is for linear

# Combining types

- Suppose that we want both $a$ and $b$ linear in process

$$a?().b![] \mid a![]$$

- We know

input

output

$$a : l_?[], b : l_![] \vdash a?(x).b![x] \qquad\qquad a : l_![] \vdash a![]$$

- We need to combine the two types for $a$

$$a : l_\sharp[], b : l_![] \vdash a?(x).b![x] \mid a![]$$

input/output

# Combining typing environments

- Combination of types

$$
\begin{array}{rcll}
l_?[\vec{T}] \uplus l_![\vec{T}] & \stackrel{\text{def}}{=} & l_\sharp[\vec{T}] & \\
T \uplus T & \stackrel{\text{def}}{=} & T & \text{if } T \text{ is not a linear type} \\
T \uplus S & \stackrel{\text{def}}{=} & \text{undefined} & \text{otherwise}
\end{array}
$$

- Combination of typing environments

$$
(\Gamma_1 \uplus \Gamma_2)(x) \stackrel{\text{def}}{=}
\begin{cases}
\Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ def'd} \\
\Gamma_1(x) & \text{if } \Gamma_1(x) \text{ def'd, } \Gamma_2(x) \text{ undef'd} \\
\Gamma_2(x) & \text{if } \Gamma_2(x) \text{ def'd, } \Gamma_2(x) \text{ undef'd} \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

# Typing System

- The rule for parallel composition

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \uplus \Gamma_2 \vdash P_1 \mid P_2}$$

- Compare with the "old" rule

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2}$$

- The typing environment is split in two, rather than reused in both branches

# Typing System - more rules

- Rules for input and for output

$$\frac{\Gamma_1 \vdash x\colon m[\vec{T}] \quad \Gamma_2, \vec{y}\colon \vec{T} \vdash P \quad m \in \{?, l_?\}}{\Gamma_1 \uplus \Gamma_2 \vdash x?(\vec{y}).P}$$

$$\frac{\Gamma_1 \vdash x\colon m[\vec{T}] \quad \Gamma_2 \vdash \vec{y}\colon \vec{T} \quad \Gamma_3 \vdash P \quad m \in \{!, l_!\}}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash x![\vec{y}].P}$$

- Rules for inaction and for values

$$\frac{\Gamma \text{ contains no linear type}}{\Gamma \vdash \mathbf{0}} \qquad \frac{}{\Gamma \vdash \mathbf{0}}$$

$$\frac{\Gamma \text{ contains no linear type}}{\Gamma, x\colon T \vdash x\colon T} \qquad \frac{}{\Gamma, x\colon T \vdash x\colon T}$$

# A good Lock Manager's client

- Good clients release the lock

$$(\upsilon s)(\text{aquireLock}![s].\ s?(\text{done}).\ \text{done}![])$$

- The expected types, as seen from the client's perspective

$$\begin{aligned} \text{done} &: l_![] \\ \text{s} &: l_?[l_![]] \\ \text{acquireLock} &: ![l_?[l_![]]] \end{aligned}$$

- Exercise: write the typing derivation

# Not all clients to the Lock Manager are typable

- A client that does not release the lock

$$(\upsilon s)(\text{aquireLock}![s].\ s?(\text{done}).\ \mathbf{0})$$

is not typable because

$$\text{done}\colon l_{!}[] \nvdash \mathbf{0}$$

- A client that releases the lock twice

$$(\upsilon s)(\text{aquireLock}![s].\ s?(\text{done}).\ (\text{done}![]\ \mid\ \text{done}![]))$$

is not typable because

$$\text{done}\colon l_{!}[] \nvdash \text{done}![] \mid \text{done}![]$$

# Session types

# Remember ftp?

- A client that uploads a file on an ftp server f

start a session

x is the session identifier

```
request f(x).
  x!["vv", 1313].
  x>{sorry: 0,
    welcome:
      x<put.
      x![myFile].
      x<quit
   }
```

authenticate

branch on the result

select operation; send argument

select operation

# The server side

- A simple ftp server

y is the
session identifier

Ftpd(f) =
  **accept** f(y).
  y?(userid, passwd).
  **if** ...
  **then** y<sorry. Ftpd[f]
  **else** y<welcome. Actions[y]

start a session
(the server's side)

select an operation
on the client

Actions(y) = y>{
  get: ... Actions[y],
  put: y?(aFile). ... Actions[y],
  quit: **0**}

branch

# Distinguish names from channels

- Names are shared among any number of partners, and used to start sessions
- There is one channel per session; channels are shared by exactly two partners, and are used for continuous interactions
- Operations on channels include
  - data transmission (input and output)
  - offer a menu (branch); pick a choice in a menu (select)

# Changes to the syntax

- Names (and variables) are $x$,$y$,$z$ as before; runtime channels are $\kappa, \kappa', \kappa''$

- Channel expressions

one end

$$k \;::=\; x \;\mid\; \kappa^+ \;\mid\; \kappa^-$$

the other end

- New process constructors

$$\begin{array}{rcl} P & ::= & \textsf{request}\; a(x).P \\ & \mid & \textsf{accept}\; a(x).P \\ & \mid & k \lhd l.P \\ & \mid & k \rhd \{l_1 : P_1, \ldots, l_n : P_n\} \\ & \mid & k?(\vec{x}).P \\ & \mid & k![\vec{x}].P \\ & \mid & \ldots \end{array}$$

was $y?(x).P$

input/output now
only within sessions

# New rules in the operational semantics

- Start a session

$$\text{accept } a(x).P \mid \text{request } a(y).Q \;\rightarrow\; (\nu\kappa)(P[\kappa^+/x] \mid Q[\kappa^-/y])$$

- Select a branch

$$\kappa^+ \lhd l_i.P \mid \kappa^- \rhd \{l_1 : Q_1, \ldots, l_n : Q_n\} \;\rightarrow\; P \mid Q_i$$

There is another rule with + and - reversed

# The type of the FTP channel as seen by the client

- The type of channel $x$

```
request f(x).
  x![“vv”, 1313].
  x>{sorry: 0,
     welcome:
        x<put.
        x![myFile].
        x<quit
     }
```

```
![String, Int].
&{sorry: End,
  welcome: Loop}
Loop = +{
  put: ![File]. Loop,
  get: ...,
  quit: End}
```

# Sorts and Types

- Distinguish value types (called sorts) from channel types (called types)
- Sorts for basic values and names

$$S \; ::= \; \mathsf{Int} \;\mid\; \mathsf{String} \;\mid\; \langle T \rangle$$

A name capable of starting a session of type $T$

- Types for channels

$$T \; ::= \; +\{l_1 : T_1, \ldots, l_n : T_n\} \;\mid\; \&\{l_1 : T_1, \ldots, l_n : T_n\}$$
$$\mid\; ![\vec{S}] \;\mid\; ?[\vec{S}] \;\mid\; \mathsf{End}$$

was $![\vec{T}]$

# The type of the FTP channel as seen by the server

- The type of channel *y*

```
Ftpd(f) =
  accept f(y).
  y?(userid, passwd).
  if ...
  then y<sorry. Ftpd[f]
  else y<welcome. Actions[y]
Actions(y) = y>{
  get: ... Actions[y],
  put: y?(aFile). ... Actions[y],
  quit: 0}
```

```
?[String, Int].
+{sorry: End,
  welcome: Loop'}

Loop' = &{
  put: ?[File]. Loop',
  get: ...,
  quit: End}
```

# Two views on the type of the FTP channel

$$
\begin{array}{ll}
![\text{String}, \text{Int}]. & ?[\text{String}, \text{Int}]. \\
\&\{\text{sorry}: \textbf{End}, & +\{\text{sorry}: \textbf{End}, \\
\quad \text{welcome}: \text{Loop}\} & \quad \text{welcome}: \text{Loop}'\} \\
\text{Loop} = +\{ & \text{Loop}' = \&\{ \\
\quad \text{put}: ![\text{File}].\ \text{Loop}, & \quad \text{put}: ?[\text{File}].\ \text{Loop}', \\
\quad \text{get}: \ldots, & \quad \text{get}: \ldots, \\
\quad \text{quit}: \textbf{End}\} & \quad \text{quit}: \textbf{End}\}
\end{array}
$$

- One says $!$, the other $?$; one says $+$, the other $\&$; one says End, the other End
- The two types are dual; the dual of $T$ is written $\overline{T}$

# Typing system

- Sequents

the classical part

the linear part

$$\Gamma \vdash P \colon \Delta$$

name: sort

channel: type

- Creating a session

accept gets one type

$$\frac{\Gamma \vdash P \colon \Delta, x \colon T}{\Gamma, a \colon \langle T \rangle \vdash \textsf{accept}\ a(x)\ \textsf{in}\ P \colon \Delta}$$

request gets the dual

$$\frac{\Gamma \vdash P \colon \Delta, x \colon \overline{T}}{\Gamma, a \colon \langle T \rangle \vdash \textsf{request}\ a(x)\ \textsf{in}\ P \colon \Delta}$$

# More rules

the arguments are classical

the channel is linear

- Send and receive

$$\frac{\Gamma \vdash \vec{x} \colon \vec{S} \qquad \Gamma \vdash P \triangleright \Delta, k \colon T}{\Gamma \vdash k![\vec{x}].P \triangleright \Delta, k \colon ![\vec{S}].T}$$

$$\frac{\Gamma, \vec{x} \colon \vec{S} \vdash P \triangleright \Delta, k \colon T}{\Gamma \vdash k?(\vec{x}).P \triangleright \Delta, k \colon ?[\vec{S}].T}$$

- Branch and select

$$\frac{\Gamma \vdash P_1 \triangleright \Delta, k \colon T_1 \qquad \ldots \qquad \Gamma \vdash P_n \triangleright \Delta, k \colon T_n}{\Gamma \vdash k > \{l_1 : P_1, \ldots, l_n : P_n\} \triangleright \Delta, k \colon \ \triangleright \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, k \colon T_j}{\Gamma \vdash k < l_j \triangleright \Delta, k \colon \ \triangleleft \{l_1 : T_1, \ldots, l_n : T_n\}}$$

# Parallel composition and name restriction

- Parallel composition and name restriction

$$\frac{\Gamma \vdash P : \Delta \qquad \Gamma \vdash Q : \Theta}{\Gamma \vdash P \mid Q : \Delta, \Theta}$$

$$\frac{\Gamma \vdash P : \Delta, k^{+} : T, k^{-} : \overline{T}}{\Gamma \vdash (\nu k)P : \Delta}$$

- A channel $k$ can only be restricted if its the types of its two ends, $k$+ and $k$-, are dual

- Subject Reduction and Type Safety hold only for balanced environments: where the two ends of each channel are of dual types

# Increasing the throughput of the FTP server

Ftpd(f) = (υt)(Loop[f,t] | Thread[t] | ... | Thread[t])

Loop(f,t) = **accept** f(y). **request** t(z). z![y]. Loop[f,t]

Thread(t) = **accept** t(w). w?(userid, pa

sending channels on channels

**if** ...

**then** w<sorry. Thread[t]

**else** w<welcome. Actions[t,w]

Actions(t,w) = w>{

get: ... Actions[t,w],

put: w?(aFile). ... Actions[t,w],

quit: Thread[t]}

# New types

- The client does not notice the difference ⇒ the type $\mathrm{T}$ of the FTP channel $y$ remains unchanged

$$\mathrm{T} = ![\mathrm{String}, \mathrm{Int}].\ \&\{\mathrm{sorry}\colon \mathbf{End}, \mathrm{welcome}\colon \mathrm{Loop}\}$$

- The type for channel $z$ is however new

sending channels on channels

$$\begin{aligned}
&\mathrm{Loop}(f,t) = \mathbf{accept}\ f(y).\\
&\quad \mathbf{request}\ t(z).\\
&\quad z![y]. \qquad\qquad\qquad\qquad z\colon ![\mathrm{T}].\ \mathbf{End}\\
&\quad \mathrm{Loop}[f,t]
\end{aligned}$$

# Channels: send and forget

- If Loop uses channel y after sending

Loop(f,t) = **accept** f(y).
    **request** t(z).
    z![y].
    y?(userid, passwd).
    ...

Thread(t) = **accept** t(w).
    w?(userid, passwd).
    ...

we will end up with three threads trying to communicate on the channel:

- The client is writing
- Loop (y) as well as one of the Threads (w) are reading ⇒ Error!

# New types, new rules

- New types for channels

$$T \;::=\; \mathsf{o}[\vec{T}] \;\mid\; \mathsf{i}[\vec{T}] \;\mid\; \mathsf{o}[\vec{S}] \;\mid\; \mathsf{i}[\vec{S}] \;\mid\; \ldots$$

- New rules for send/receive channel

Process $P$ cannot use $k'$ anymore

$$\frac{\Gamma \vdash P \triangleright \Delta, k \colon T}{\Gamma \vdash k![k'].P \triangleright \Delta, k \colon ![T'].T, k' : T'}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, k \colon T, k' : T'}{\Gamma \vdash k!(k').P \triangleright \Delta, k \colon ![T'].T}$$

# Further systems

- Recursive types
- Various forms of receptiveness
- Polymorphism
- Type systems for deadlock freedom