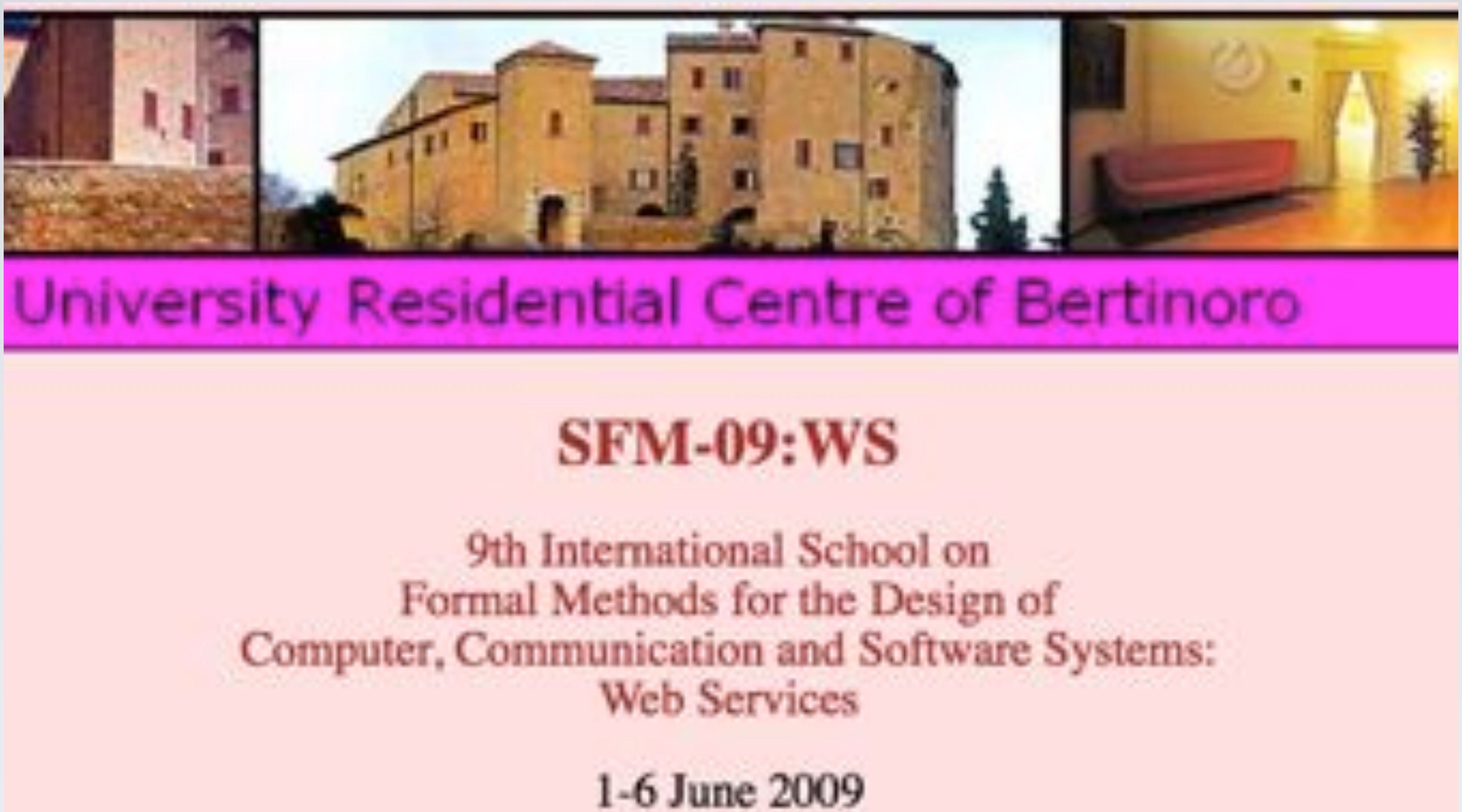
SESSIONS, FROM TYPES TO PROGRAMMING LANGUAGES

Vasco T Vasco
Universidade Lisboa

Behavioural Types Workshop
19 April 2011



FUNDAMENTALS OF SESSION TYPES

LANGUAGE PRIMITIVES AND TYPE DISCIPLINE FOR STRUCTURED COMMUNICATION-BASED PROGRAMMING

KOHEI HONDA*, VASCO T. VASCONCELOS†, AND MAKOTO KUBO‡

2.2. **Syntax Summary.** We summarise the syntax we have introduced so far. Base sets are: *names*, ranged over by $a, b, \ldots$; *channels*, ranged over by k, k'; *variables*, ranged over by $x, y, \ldots$; *constants* (including names, integers and booleans), ranged over by

THERE ARE NAMES AND THERE ARE CHANNELS

LANGUAGE PRIMITIVES AND TYPE DISCIPLINE FOR STRUCTURED COMMUNICATION-BASED PROGRAMMING

KOHEI HONDA*, VASCO T. VASCONCELOS†, AND MAKOTO KUBO‡

Definition 5.1 (Types). Given *type variables* $(t, t', \ldots)$ and *sort variables* $(s, s', \ldots)$, *sorts* $(S, S', \ldots)$ and *types* $(\alpha, \beta, \ldots)$ are defined by the following grammar.

$$
\begin{aligned}
S &::= \mathbf{nat} \mid \mathbf{bool} \mid \langle \alpha, \overline{\alpha} \rangle \mid s \mid \mu s.S \\
\alpha &::= \downarrow[\tilde{S}];\alpha \mid \downarrow[\alpha];\beta \mid \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid \mathbf{1} \mid \perp \\
&\quad \mid \uparrow[\tilde{S}];\alpha \mid \uparrow[\alpha];\beta \mid \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid t \mid \mu t.\alpha
\end{aligned}
$$

AND THERE ARE SORTS AND THERE ARE TYPES

A Calculus of Mobile Processes, Part I

Robin Milner, University of Edinburgh, Scotland
Joachim Parrow, Swedish Institute of Computer Science, Kista, Sweden
David Walker, University of Technology, Sydney, Australia

June 1989 (Revised September 1990)

2 The calculus

We presuppose an infinite set $\mathcal{N}$ of *names*, and let u, v, w, x, y, z range over names. We also presuppose a set $\mathcal{K}$ of *agent identifiers*, each with an *arity* – an integer ≥ 0. We let $A, B, C, \ldots$ range over agent identifiers. We now let $P, Q, R, \ldots$ range over the *agents* or *process expressions*, which are of six kinds as follows:

IN THE PI CALCULUS THERE ARE ONLY NAMES

Acta Informatica (2005)
DOI 10.1007/s00236-005-0177-z

ORIGINAL ARTICLE

Simon Gay · Malcolm Hole

Subtyping for session types in the pi calculus

Session types

$$
\begin{array}{rcll}
S & ::= & X & \textit{type variable} \\
 & | & \mathsf{end} & \textit{terminated session} \\
 & | & ?[T_1, \ldots, T_n].S & \textit{input} \\
 & | & ![T_1, \ldots, T_n].S & \textit{output} \\
 & | & \&\langle l_1 : S_1, \ldots, l_n : S_n \rangle & \textit{branch} \\
 & | & \oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle & \textit{choice} \\
 & | & \mu X.S & \textit{recursive session type}
\end{array}
$$

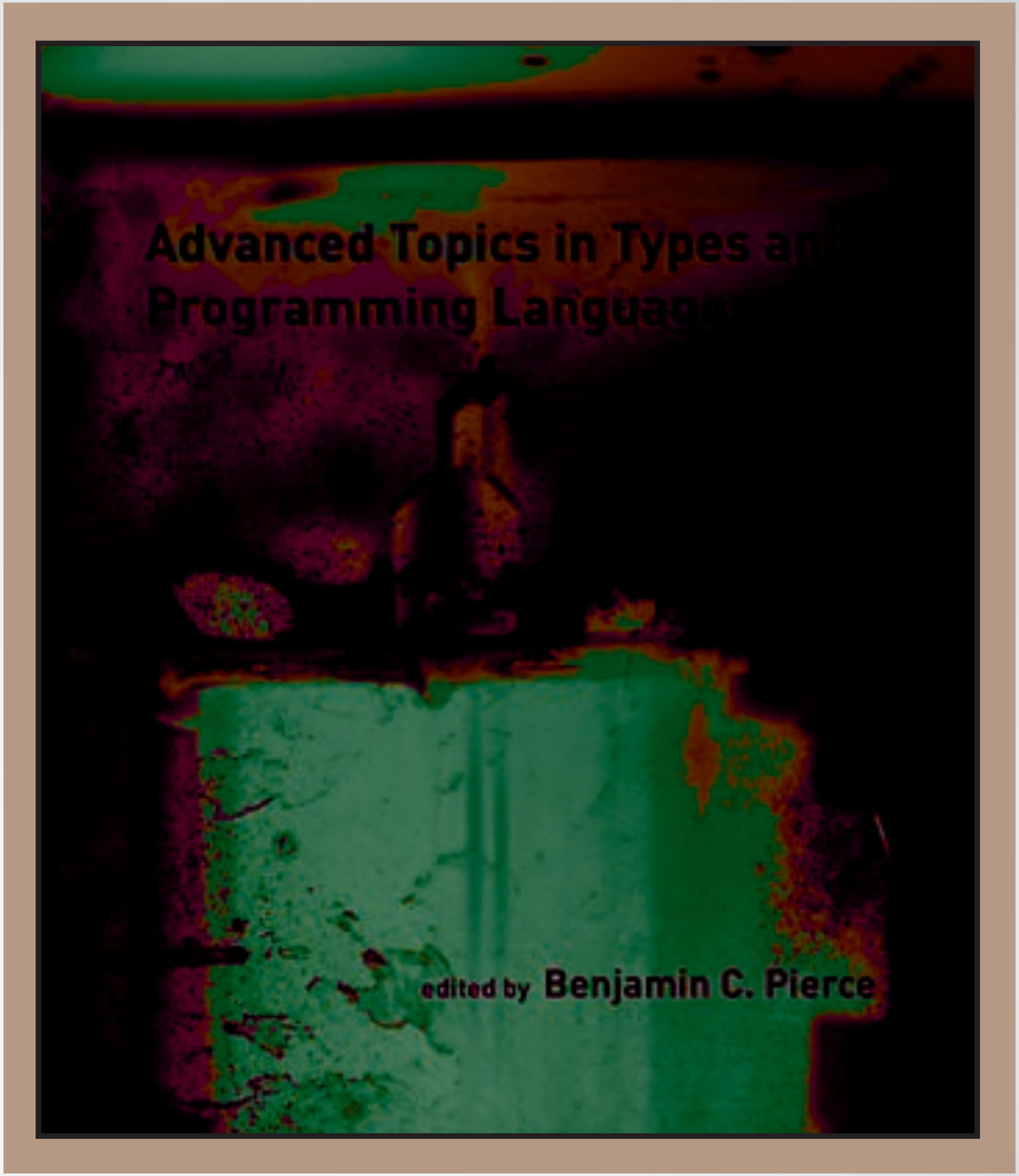
Types

$$
\begin{array}{rcll}
T & ::= & X & \textit{type variable} \\
 & | & S & \textit{session type} \\
 & | & \hat{}[T_1, \ldots, T_n] & \textit{standard channel type} \\
 & | & \mu X.T & \textit{recursive channel type}
\end{array}
$$

Fig. 1 Syntax of types

BUT WE STILL HAVE TYPES AND SESSION TYPES

THEN I (RE)READ THE BROWN BOOK...



1 *Substructural Type Systems*

David Walker

Advanced type systems make it possible to restrict access to data structures and to limit the use of newly-defined operations. Oftentimes, this sort of access control is achieved through the definition of new abstract types under control of a particular module. For example, consider the following simplified file system interface.

...FROM CHAPTER ONE

AND I LEARNT...

- Resources can be either linear or shared (resources are values in a call be value lambda calculus)
- Resources are heap allocated; linear resources are deallocated after being used
- Access to resources is mediated by types, which can be
  - qualified as linear or unrestricted (shared)
- The main result says that there are no dangling pointers to the heap, from a well typed process

TRANSPOSING TO THE PI CALCULUS

- Names can be either linear or shared (you can call them channels if you like)
- Resources (channels, input processes) are either linear or shared; linear resources are deallocated after being used
- Access to resources is mediated by types, which can be
  - qualified as linear or unrestricted (shared)
- The main result states that if a thread reads on a channel end, the other writes on the other end

q	$::=$		*Qualifiers:*
		lin	linear
		un	unrestricted
p	$::=$		*Pretypes:*
		unit	unit
		end	termination
		$?T.T$	receive
		$!T.T$	send
		$\oplus\{l_i : T_i\}_{i \in I}$	select
		$\&\{l_i : T_i\}_{i \in I}$	branch
T	$::=$		*Types:*
		$q\ p$	qualified pretype
		a	type variable
		$\mu a.T$	recursive type

Figure 1: The syntax of types

JUST TYPES... LIN/UN ANNOTATED

WHAT NEW THINGS CAN WE DO WITH THESE TYPES?

1 *Substructural Type Systems*

David Walker

Typing

To ensure that linear objects are used exactly once, our type system maintains two important invariants.

1. Linear variables are used exactly once along every control-flow path.
2. Unrestricted data structures may not contain linear data structures. More generally, data structures with less restrictive type may not contain data structures with more restrictive type.

LINEAR CBV LAMBDA

- An unrestricted channel cannot evolve into a linear channel,

  but...

- A linear channel may evolve into an unrestricted channel (this is new)

TRANSPOSING TO THE PI CALCULUS

AN ONLINE PETITION SERVER

HOW IT WORKS

- Petition writers start by providing the title of the petition, a piece of text describing the situation and what is needed, and the deadline for signature collection. The service allows this information to be added with no particular order; writers can even resubmit information if needed.
- Once writers are happy with the petition details, they commit to the uploaded data and seek approval for starting the petition. If accepted, writers may start promoting the petition.
- Promoting a petition means two things: signing and disseminating. The petition writer may sign the petition, and must get people to sign it, by letting them know of the newly created petition.

THE TYPE OF THE ONLINE PETITION IN OUR SYNTAX

Petition = **lin**⊕{*setTitle*: **lin**!string.Petition, *setDate*: **lin**!date.Petition,
submit: **lin**&{*accepted*: Promotion, *denied*: **lin**?string.**lin end**}}
Promotion = **un**!string.Promotion

- Promotion = **un**!string.Promotion abbreviated to *!string
- This is the only sort of interesting un type, apart from **un end**

SOME PI CODE

```
SaveTheWolf :: ∗?Petition
SaveTheWolf ps =
  ps?p.
    p ◁ setDate. p!(31,12,2010).
    p ◁ setTitle. p!"Save the Wolf".
    p ◁ setDate. p!(31,12,2100).
    p ◁ submit.
    p ▷ {accepted:
          Signatory1 p |
          Signatory2 p |
          p!"me"
        denied:
          p?x.
          close p
      }
Signatory1 :: ∗!string
Signatory1 p =
  p!"signatory1"
Signatory2 :: ∗!string
Signatory2 p =
  Signatory3 p | p!"signatory2"
Signatory3 :: ∗!string
Signatory3 p =
  inaction
```

```
Server :: ∗!Petition
Server ps =
    (new p1 p2)
    ps!p2.(
      Setup (p1,(1,1,1970),"Save me
      Server ps)
Setup :: Petition ∗ date ∗ string
Setup (p, d, t) =
  p ▷ {setDate: p?d'.Setup (p, d', t),
      setTitle: p?t'.Setup (p, d, t'),
      submit: p ◁ accepted.
              Promotion (p, [])
      }
Promotion :: ∗?string ∗ stringList
Promotion (p, l) =
    p?s.Promotion (p, s :: l)
```

```
Main =
  (new ps1 ps2)
  Server ps1 |
  SaveTheWolf ps2
```

TYPING CONTEXTS

- Programs are typed against a context describing the types for the free identifiers. Typing contexts are finite maps Γ from identifiers to types.

- When type checking processes with two sub-processes we pass the unrestricted part of the context to both processes, while splitting the linear part in two and passing a different part to each process.

1 *Substructural Type Systems*

David Walker

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\varnothing = \varnothing \circ \varnothing \quad \text{(M-Empty)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}un}\ \mathsf{P} = (\Gamma_1, \mathtt{x{:}un}\ \mathsf{P}) \circ (\Gamma_2, \mathtt{x{:}un}\ \mathsf{P})} \quad \text{(M-Un)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}lin}\ \mathsf{P} = (\Gamma_1, \mathtt{x{:}lin}\ \mathsf{P}) \circ \Gamma_2} \quad \text{(M-Lin1)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}lin}\ \mathsf{P} = \Gamma_1 \circ (\Gamma_2, \mathtt{x{:}lin}\ \mathsf{P})} \quad \text{(M-Lin2)}$$

Figure 1-4: Linear lambda calculus: Context splitting

CONTEXT SPLITTING

ALL TYPING RULES

Typing rules for values

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash ()\colon q\,\mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T} \qquad \text{(T-UNIT,T-VAR)}$$

Typing rules for processes

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{inaction}} \qquad \frac{\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2} \qquad \frac{\Gamma, x\colon T, y\colon \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P}$$

$$\text{(T-INACT,T-PAR,T-RES)}$$

$$\frac{\Gamma_1 \vdash x\colon q\,!T_1.T_2 \qquad \Gamma_2 \vdash v\colon T_1 \qquad \Gamma_3 + x\colon T_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!v.P} \qquad \text{(T-OUT)}$$

$$\frac{\Gamma_1 \vdash x\colon q\,?T_1.T_2 \qquad (\Gamma_2, y\colon T_1) + x\colon T_2 \vdash P \qquad q(\Gamma_2)}{\Gamma_1 \circ \Gamma_2 \vdash q\,x?y.P} \qquad \text{(T-IN)}$$

$$\frac{\Gamma \text{ completed}}{\Gamma \vdash \mathbf{0}}\ \text{T-Nil} \qquad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}\ \text{T-Par} \qquad \frac{\Gamma \vdash P \quad \Gamma \text{ unlimited}}{\Gamma \vdash !P}\ \text{T-Rep}$$

$$\frac{\Gamma, x:T \vdash P \qquad T \text{ is not a session type}}{\Gamma \vdash (\nu x:T)P}\ \text{T-New}$$

$$\frac{\Gamma, x^+:S, x^-:S' \vdash P \qquad S \perp_c S'}{\Gamma \vdash (\nu x:S)P}\ \text{T-NewS}$$

$$\frac{\Gamma, x^p:S, \tilde{y}:\tilde{U} \vdash P \qquad \tilde{T} \leqslant_c \tilde{U}}{\Gamma, x^p:?[\tilde{T}].S \vdash x^p?[\tilde{y}:\tilde{U}].P}\ \text{T-InS} \qquad \frac{\Gamma, x^p:S \vdash P \qquad \tilde{U} \leqslant_c \tilde{T}}{(\Gamma, x^p:![\tilde{T}].S) + \tilde{y}^{\tilde{q}}:\tilde{U} \vdash x^p![\tilde{y}^{\tilde{q}}].P}\ \text{T-OutS}$$

$$\frac{\Gamma, x:\hat{}[\tilde{T}], \tilde{y}:\tilde{U} \vdash P \qquad \tilde{T} \leqslant_c \tilde{U}}{\Gamma, x:\hat{}[\tilde{T}] \vdash x?[\tilde{y}:\tilde{U}].P}\ \text{T-In} \qquad \frac{\Gamma, x:\hat{}[\tilde{T}] \vdash P \qquad \tilde{U} \leqslant_c \tilde{T}}{(\Gamma, x:\hat{}[\tilde{T}]) + \tilde{y}^{\tilde{q}}:\tilde{U} \vdash x![\tilde{y}^{\tilde{q}}].P}\ \text{T-Out}$$

$$\frac{m \leqslant n \quad \forall i \in \{1,\ldots,m\}.(\Gamma, x^p:T_i \vdash P_i)}{\Gamma, x^p:\&\langle l_i : T_i \rangle_{1\leqslant i\leqslant m} \vdash x^p \triangleright \{ l_i : P_i \}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

$$\frac{l = l_i \in \{l_1,\ldots,l_n\} \quad \Gamma, x^p:T_i \vdash P}{\Gamma, x^p:\oplus\langle l_i : T_i \rangle_{1\leqslant i\leqslant n} \vdash x^p \triangleleft l.P}\ \text{T-Choose}$$

Fig. 9 Typing rules

C
O
N
T
R
A
S
T

SESSION TYPES IN AN OBJECT-BASED LANGUAGE

- In channel-based languages, processes communicate by exchanging messages on (session governed) channels
- In our object-oriented language threads communicate solely by calling methods on (session governed) object references
- This is in clear contrast with more conventional approaches that add communication channels to an object-oriented language

CHANNEL OPERATIONS AS OO CONCEPTS

- A **selection** operation (previously identified with a left triangle, ▹) is identified with a method call
- An **output** operation can only be identified with argument passing within a method call
- An **input** operation can only occur as the result of a method call
- What about **branching**? How can a target object force a branch on a client? For a simple binary branch, boolean methods force such a test, via conditional expressions. For more general branching structures we use conventional enumerations (enum) and a switch construct.

ANNOTATE CLASSES WITH A SESSION TYPE

```
class Petition {
  usage Setup where
  Setup = lin&{setTitle: Setup,
        setDate: Setup,
        submit: lin⊕{accepted: Promotion,
                     denied: lin end}}
  Promotion = un&{sign: Promotion,
                  howMany: Promotion};
```

SOME MOOL CODE

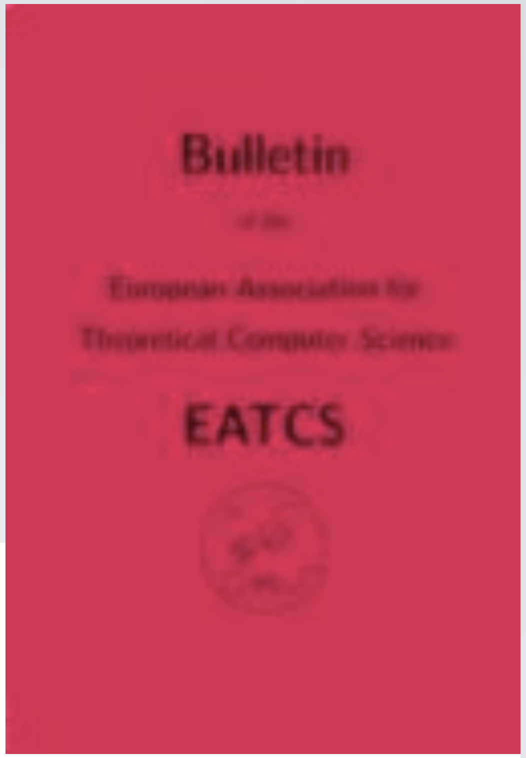
```
class PetitionServer {
   Petition newPetition() {new Petition ();}
}
class Petition {
   usage Setup where
   Setup = lin&{setTitle: Setup,
          setDate: Setup,
          submit: lin⊕{accepted: Promotion,
                        denied: lin end}}
   Promotion = un&{sign: Promotion,
                   howMany: Promotion};
   string title = "Save me";
   Date date = new Date(1,1,1970);
   List signatures = new List();
   unit setTitle(string t) { title = t; }
   unit setDate(string d) { date = d; }
   Answer submit() { Answer.accepted; }
   sync unit sign(string name) {
      signatures.add(name);
   }
   int howMany() { signatures.length(); }
}
```

CONCLUSION

- lin/un approach to session types opens interesting avenues (see talk by Giunti)

- Session types in pi, functional, OO languages (beatcs, feb 2011)

- Compiler for Mool available online



THAT IS IT!

DOUBLE BINDER

- The pi calculus (with free output), when considered in conjunction with session types, is known to require a means to distinguish the two ends of a session channel
- Two approaches for distinguishing the ends of a channel are available in the literature: polarized channel variables, and form of channel **double binder**
- A **new**-constructor (new x1 x2) creates a fresh channel and **two** identifiers, each describing one end of a channel

SOME MOOL CODE

```
enum Answer = {accepted, denied}
class SaveTheWolf {
  usage lin&{init: lin&{run: un end}};
  Petition p;
  Signatory[Sign] signatory1;
  Signatory[Sign] signatory2;
  unit init(PetitionServer s,
        Signatory[Sign] s1,
        Signatory[Sign] s2) {
    p = s.newPetition();
    signatory1 = s1;
    signatory2 = s2;
  }
  unit run() {
    p.setDate
      (new Date(31, 12, 2010));
    p.setTitle("Save the Wolf");
    p.setDate
      (new Date(31, 12, 2100));
    switch (p.submit()) {
      case Answer.accepted:
        fork signatory1.signPlease(p);
        fork signatory2.signPlease(p);
        p.sign("me");
      case Answer.denied:
        free p;
    }
  }
}
```

```
class Main {
  unit main() {
    PetitionServer server =
      new PetitionServer();
    Signatory s1 = new Signatory();
    s1.setName ("signatory1");
    Signatory s2 = new Signatory();
    s2.setName ("signatory2");
    SaveTheWolf wolf =
      new SaveTheWolf();
    wolf.init(server, s1, s2);
    fork wolf.run();
  }
}
```

```
class Signatory {
  usage lin&{setName: Sign} where
  Sign = un&{signPlease: Sign};
  string name;
  unit setName(string n) {
    name = n;
  }
  unit signPlease
    ( Petition [Promotion] p) {
    p.sign (name);
```