SESSIONS, FROM TYPES TO PROGRAMMING LANGUAGES

Vasco T Vasco
Universidade Lisboa

Behavioural Types Workshop
19 April 2011
FUNDAMENTALS OF SESSION TYPES
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 $x, u, \ldots$
- *constants* (including *names*, integers and booleans), ranged over by $v$.

THERE ARE NAMES AND THERE ARE CHANNELS
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.

\[
S ::= \text{nat} \mid \text{bool} \mid \langle \alpha, \overline{\alpha} \rangle \mid s \mid \mu s.S
\]

\[
\alpha ::= \↓[\tilde{S}]; \alpha \mid \↓[\alpha]; \beta \mid \&\{l_1: \alpha_1, \ldots, l_n: \alpha_n\} \mid 1 \mid \perp
\]

\[
\mid \↑[\tilde{S}]; \alpha \mid \↑[\alpha]; \beta \mid \oplus\{l_1: \alpha_1, \ldots, l_n: \alpha_n\} \mid t \mid \mu t.\alpha
\]

AND THERE ARE SORTS AND THERE ARE TYPES
2 The calculus

We presuppose an infinite set $\mathcal{N}$ of \textit{names}, and let $u, v, w, x, y, z$ range over names. We also presuppose a set $\mathcal{K}$ of \textit{agent identifiers}, each with an \textit{arity} – an integer $\geq 0$. We let $A, B, C, \ldots$ range over agent identifiers. We now let $P, Q, R, \ldots$ range over the \textit{agents} or \textit{process expressions}, which are of six kinds as follows:
Extending the pi calculus with the session types proposed by Honda et al. allows high-level specifications of structured patterns of communication, such as client-server protocols, to be expressed as types and verified by static type-checking. We define a notion of subtyping for session types, which allows protocol specifications to be extended in order to describe richer behaviour; for example, an implemented server can be refined without invalidating type-correctness of an overall system. We formalize the syntax, operational semantics and typing rules of an extended pi calculus, prove that typability guarantees absence of run-time communication errors, and show that the typing rules can be transformed into a practical typechecking algorithm.

Fig. 1 Syntax of types

BUT WE STILL HAVE TYPES AND SESSION TYPES
The study of type systems for programming languages now touches many areas of computer science, from language design and implementation to software engineering, network security, databases, and analysis of concurrent and distributed systems. This book offers accessible introductions to key ideas in the field, with contributions by experts on each topic. The topics covered include precise type analyses, which extend simple type systems to give them a better grip on the run time behavior of systems; type systems for low-level languages; applications of types to reasoning about computer programs; type theory as a framework for the design of sophisticated module systems; and advanced techniques in ML-style type inference.

Advanced Topics in Types and Programming Languages builds on Benjamin Pierce's Types and Programming Languages (MIT Press, 2002); most of the chapters should be accessible to readers familiar with basic notations and techniques of operational semantics and type systems—the material covered in the first half of the earlier book.

Advanced Topics in Types and Programming Languages can be used in the classroom and as a resource for professionals. Most chapters include exercises, ranging in difficulty from quick comprehension checks to challenging extensions, many with solutions. Additional material can be found at <http://www.cis.upenn.edu/~bcpierce/attapl>.

Benjamin C. Pierce is Professor of Computer and Information Science at the University of Pennsylvania. He is the author of Basic Category Theory for Computer Scientists (MIT Press, 1991) and Types and Programming Languages (MIT Press, 2002).

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

```
type file
val open : string \rightarrow file option
val read : file \rightarrow string * file
val append : file * string \rightarrow file
val write : file * string \rightarrow file
val close : file \rightarrow unit
```

By declaring that the type `file` is abstract, the implementer of the module can maintain strict control over the representation of files. Consequently, the implementer may assume that the invariants that he or she establishes upon opening a file hold before any read, append, write or close.

While abstract types are a powerful means of controlling the structure of data, they are not sufficient to limit the ordering and number of uses of functions in an interface. Try as we might, there is no (static) way to prevent a file from being read after it has been closed. Likewise, we cannot stop a client from closing a file twice or forgetting to close a file.

This chapter introduces substructural type systems, which augment standard type abstraction mechanisms with the ability to control the number and order of uses of a data structure or operation. Substructural type systems are particularly useful for constraining interfaces that provide access to system...
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
Figure 1: The syntax of types

<table>
<thead>
<tr>
<th>q ::=</th>
<th>Qualifiers:</th>
<th>( !T.T )</th>
<th>send</th>
</tr>
</thead>
<tbody>
<tr>
<td>lin</td>
<td>linear</td>
<td>( \oplus {l_i : T_i}_{i \in I} )</td>
<td>select</td>
</tr>
<tr>
<td>un</td>
<td>unrestricted</td>
<td>( &amp; {l_i : T_i}_{i \in I} )</td>
<td>branch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p ::=</th>
<th>Pretypes:</th>
<th>T ::=</th>
<th>Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>unit</td>
<td>q p</td>
<td>qualified pretype</td>
</tr>
<tr>
<td>end</td>
<td>termination</td>
<td>a</td>
<td>type variable</td>
</tr>
<tr>
<td>(?T.T)</td>
<td>receive</td>
<td>( \mu a.T )</td>
<td>recursive type</td>
</tr>
</tbody>
</table>

JUST TYPES... LIN/UN ANNOTATED
WHAT NEW THINGS CAN WE DO WITH THESE TYPES?
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.
• An unrestricted channel cannot evolve into a linear channel, but...

• A linear channel may evolve into an unrestricted channel (this is new)
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 = \textbf{lin} \oplus \{setTitle: \textbf{lin}\!string.\text{Petition}, setDate: \textbf{lin}\!date.\text{Petition}, submit: \textbf{lin}\&\{accepted: \text{Promotion}, denied: \textbf{lin}\?string.\textbf{lin}\textend\}\}\}

Promotion = \textbf{un}\!string.\text{Promotion}

- Promotion = \textbf{un}\!string.\text{Promotion} abbreviated to *!string
- This is the only sort of interesting \textit{un} type, apart from \textbf{un end}
SaveTheWolf :: Petition

Server :: Petition

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

Figure 4: Petition example in the pi–calculus

...
TYPING CONTEXTS

• Programs are typed against a context describing the types for the free identifiers. Typing contexts are finite maps \( \Gamma \) 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.
Context Split

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset = \emptyset \circ \emptyset$</td>
<td>(M-EMPTY) $\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\Gamma = \Gamma_1 \circ \Gamma_2$</td>
<td>(M-UN) $\Gamma_1 \circ \Gamma_2$</td>
<td>$\Gamma_1 \circ \Gamma_2$</td>
</tr>
<tr>
<td>$\Gamma, x:\text{un} P = (\Gamma_1, x:\text{un} P) \circ (\Gamma_2, x:\text{un} P)$</td>
<td>(M-LIN1) $\Gamma_1 \circ \Gamma_2$</td>
<td>$\Gamma_1 \circ \Gamma_2$</td>
</tr>
<tr>
<td>$\Gamma, x:\text{lin} P = (\Gamma_1, x:\text{lin} P) \circ \Gamma_2$</td>
<td>(M-LIN2) $\Gamma_1 \circ (\Gamma_2, x:\text{lin} P)$</td>
<td>$\Gamma_1 \circ (\Gamma_2, x:\text{lin} P)$</td>
</tr>
</tbody>
</table>

Figure 1-4: Linear lambda calculus: Context splitting
ALL TYPING RULES

Typing rules for values

\[
\begin{align*}
\text{un}(\Gamma) & \quad \Rightarrow \quad \Gamma \vdash () : q \text{ unit} \\
\text{un}(\Gamma) & \quad \Rightarrow \quad \Gamma, x : T \vdash x : T
\end{align*}
\]

(T-UNIT, T-VAR)

Typing rules for processes

\[
\begin{align*}
\text{un}(\Gamma) & \quad \Rightarrow \quad \Gamma \vdash \text{inaction} \\
\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2 & \quad \Rightarrow \quad \Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2 \\
\Gamma, x : T, y : \overline{T} & \vdash P \\
\Gamma_1 \vdash x : q ! T_1.T_2 & \quad \Gamma_2 \vdash v : T_1 & \quad \Gamma_3 + x : T_2 \vdash P \\
\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 & \vdash x ! v.P \\
\Gamma_1 \vdash x : q ? T_1.T_2 & \quad (\Gamma_2, y : T_1) + x : T_2 \vdash P & \quad q(\Gamma_2) \\
\Gamma_1 \circ \Gamma_2 & \vdash q x ? y.P
\end{align*}
\]

(T-INACT, T-PAR, T-RES)

(T-OUT)

(T-IN)

Figure 1: Typing rules for the pi-calculus
Fig. 9 Typing rules
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.
class Petition {
    usage Setup where
    Setup = lin&{setTitle: Setup,
                setDate: Setup,
                submit: lin⊕{accepted: Promotion,
                               denied: lin end}}
    Promotion = un&{sign: Promotion,
                    howMany: Promotion};
```plaintext
1  class PetitionServer {  
2      Petition newPetition() { new Petition(); }  
3  }  
4  class Petition {  
5      usage Setup where  
6          Setup = lin\&{setTitle: Setup,  
7              setDate: Setup,  
8              submit: lin\oplus{accepted: Promotion,  
9                  denied: lin end}}}  
10     Promotion = un\&{sign: Promotion,  
11                     howMany: Promotion};  
12     string title = "Save me";  
13     Date date = new Date(1,1,1970);  
14     List signatures = new List();  
15     unit setTitle(string t) { title = t; }  
16     unit setDate(string d) { date = d; }  
17     Answer submit() { Answer.accepted; }  
18     sync unit sign(string name) {  
19         signatures.add(name);  
20     }  
21     int howMany() { signatures.length(); }  
22  }
```
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!
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.
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;
        }
    }