Multiparty Asynchronous Session Types

Tony Garnock-Jones <tonyg@ccs.neu.edu>

April 3, 2012

1 π-calculus: Communication, the ultimate

See [Milner(1991)] for a good, though early, introduction to π-calculus. Briefly,

$$\begin{array}{rcl} P, Q & ::= & 0 \\ & | & x!v.P \\ & | & x?y.P \\ & | & \nu x.P \\ & | & P|Q \\ v & ::= & x \end{array}$$

For brevity, we write $x!v$ for $x!v.0$, and $x!.P$ and $x?.P$ when we only want to communicate for a synchronisation effect.

We can replace $x!v.P$ with $x!e.P$, including some expressions e, and then we end up with something a little Erlang-like, with separate calculation and communication language fragments. Another interesting direction to take this idea is the *join calculus* (a lovely development of which is given in the first section of [Fournet and Gonthier(2000)]).

Structural congruence is a crucial component of the semantics of π:

$$\begin{array}{rcll} \nu x.0 & \equiv & 0 & \\ P|Q & \equiv & Q|P & \\ \nu x.\nu y.P & \equiv & \nu y.\nu x.P & \\ \nu x.(P|Q) & \equiv & P \,|\, \nu x.Q & (\text{if } x \notin fn(P)) \end{array}$$

That last rule lets restrictions not trapped under some pending communication "float" to the outermost position in the program. It's called a "scope extrusion" rule and it structures the topology of the communication channels connecting subprograms.

Moving on to reduction:

$$x!v.P \mid x?y.Q \;\rightarrow\; P \mid Q\{v/y\} \qquad \text{[comm]}$$

There are also structural rules:

$$\frac{P \rightarrow P'}{\nu x.P \rightarrow \nu x.P'} \qquad \text{[res]}$$

$$\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \qquad \text{[par]}$$

$$\frac{P \equiv P' \;\; P' \rightarrow Q' \;\; Q' \equiv Q}{P \rightarrow Q} \qquad \text{[struct]}$$

2 New and interesting kinds of wrongness

Let's extend π with some ground types and simple primitives, so we can do more than just passing channel-names over channels. We'll take natural numbers:

$$\begin{array}{rcl} P & ::= & \ldots \mid x!e.P \mid \ldots \\ v & ::= & x \mid \lceil n \rceil \\ e & ::= & e + e \mid \ldots \end{array}$$

Now we have similar kinds of wrongness to the λ-calculus:

$$\nu y.x!y \mid x?v.k!(v+5)$$

But we also have new kinds of wrongness from *nondeterminism* that don't appear in λ:

$$\nu y.x!y \mid x!123 \mid x?a.x?b.k!(b+5)$$

This program has two possible reduction sequences:

$$\begin{array}{rcl} \nu y.x!y \mid x!123 \mid x?a.x?b.k!(b+5) & \rightarrow & x!123 \mid x?b.k!(b+5) \\ & \rightarrow & k!(123+5) \\ & \rightarrow & k!128 \end{array}$$

and

$$\begin{array}{rcl} \nu y.x!y \mid x!123 \mid x?a.x?b.k!(b+5) & \rightarrow & \nu y.x!y \mid x?b.k!(b+5) \\ & \rightarrow & \nu y.k!(y+5) \\ & \not\rightarrow & \end{array}$$

We can also get into *data races*:

$$x!123 \mid x?a.y!a \mid x?b.z!b$$

and *deadlocks*:

$$x?a.y!123 \mid y?b.x!234$$

It'd be nice if our type systems could help us avoid such wrongness! To avoid stuckness in the expression fragment, we can introduce *port sorting* [Pierce and Sangiorgi(1996)], which will be a part of the system I'll be introducing below, but because the real π-calculus includes recursion in some form or other[1], deciding whether a program has a race or a deadlock is in general undecidable. So we make a tradeoff between the programs we can express and the properties we can prove about those programs.

3 Session Types: Typing Behaviour

In the λ-calculus, we find ourselves thinking about the types of data that we pass around. In the π-calculus, we think much more frequently in terms of the types of *behaviour* of processes we run alongside. Equivalences in λ tend to be structural; in π, behavioural.

So we find ourselves looking for types describing *behaviours.* In particular, we want to be able to think about whole conversations: about *groups* of processes interacting with each other, and about invariants of that interaction.

The type system given in [Honda et al.(2008)Honda, Yoshida, and Carbone] ensures that the following properties hold in a π-calculus variant (quoted verbatim):

[1] Either via syntax $!P$, where $!P \equiv P|!P$, or via *letrec*-style recursive definitions and process variables.

1. Interactions within a session never incur a communication error (communication safety).
2. Channels for a session are used linearly (linearity) and are deadlock-free in a single session (progress).
3. The communication sequence in a session follows the scenario declared in the session type (session fidelity, predictability).

The original work on session types ([Honda et al.(1998)Honda, Vasconcelos, and Kubo] and many others) concentrated on *binary* session types because the types of the two parties in a binary session are duals of each other. Let's take an example from [Honda et al.(2008)Honda, Yoshida, and Carbone] to show this, but first we'll define the language we're going to be using.

4 The language

Definition 1. (Processes.) This is most similar to the language given in [Mostrous et al.(2009)Mostrous, Yoshida, and Hon which is essentially that of [Honda et al.(2008)Honda, Yoshida, and Carbone] with polyadicity and session delegation removed for simplicity.

$$
\begin{array}{rcll}
P, Q & ::= & \overline{x}[2, ..., n](\tilde{s}).P & \\
& | & x[p](\tilde{s}).P & \\
& | & s!e; P & \\
& | & s?x; P & \\
& | & s \triangleleft \ell; P & \\
& | & s \triangleright \{\ell_i : P_i\}_{i \in I} & \\
& | & \textbf{if}\, e\, \textbf{then}\, P\, \textbf{else}\, Q & \\
& | & P|Q & \\
& | & \mathbf{0} & \\
& | & \nu x.P & \\
& | & \textbf{def}\, \{X_i\langle \tilde{x_i}\tilde{s_i}\rangle = P_i\}_{i \in I}\, \textbf{in}\, P & \\
& | & X\langle \tilde{e}\tilde{s}\rangle & \\
& | & s : \tilde{h} & \text{(queue)}
\end{array}
$$

$$
\begin{array}{rcl}
x, s & ::= & \text{variables} \\
p & ::= & \text{participant IDs; generally, small integers } \mathbb{N} \\
\ell & ::= & \text{labels} \\
e & ::= & \text{expressions} \\
v & ::= & \text{values} \\
h & ::= & \ell \mid v \qquad \text{(queue entries)}
\end{array}
$$

Notice the final entry in the grammar for P, $s : \tilde{h}$. This is a queue structure, buffering sent messages until they are received. The introduction of a queue for each channel makes the system asynchronous—that is, a sending action is never blocked—and order preserving.

Definition 2. (Operational Semantics.) The structural congruence from plain π is modified in a natural way. The reduction rules are modified in a more interesting way: instead of the normal comm

rule, we have rules for output, and rules for input. We also have a special multicast rule for linking and kicking off a session:

$$\text{[link]}$$
$$\overline{a}[2,...,n](\tilde{s}).P_1 \mid a[2].P_2 \mid ... \mid a[n].P_n \quad \rightarrow \quad \nu\tilde{s}.(P_1|P_2|...|P_n \mid s_1 : \emptyset|...|s_j : \emptyset) \quad (\text{where } j = |\tilde{s}|)$$

$$\text{[output]}$$
$$s!e; P \mid s : \tilde{h} \quad \rightarrow \quad P \mid s : \tilde{h} \cdot v \qquad (e \downarrow v)$$
$$s \triangleleft \ell; P \mid s : \tilde{h} \quad \rightarrow \quad P \mid s : \tilde{h} \cdot \ell$$

$$\text{[input]}$$
$$s?x; P \mid s : v \cdot \tilde{h} \quad \rightarrow \quad P\{v/x\} \mid s : \tilde{h}$$
$$s \triangleright \{\ell_i : P_i\}_{i \in I} \mid s : \ell_k \cdot \tilde{h} \quad \rightarrow \quad P_k \mid s : \tilde{h} \qquad (k \in I)$$

Example 3. (One-buyer protocol.) Seller S is selling a book. Buyer B wants to buy it. Processes for the interaction:

$$\begin{aligned} S \;=\; & x?title;\ x!quote; \\ & x \triangleright \{quit : \mathbf{0} \\ & \qquad ok : x?address;\ x!date;\ ...\} \end{aligned}$$

$$\begin{aligned} B \;=\; & x!title;\ x?quote; \\ & \mathbf{if}\ quote \geq limit \\ & \quad \mathbf{then}\ x \triangleleft quit \\ & \quad \mathbf{else}\ x \triangleleft ok;\ x!address;\ x?date;\ ... \end{aligned}$$

Note that there's just the one channel involved, x, because there are only two participants. The *binary* session type for this protocol from the seller's point of view is

$$?string\ !int\ \&\{quit : end,\ ok :?string\ !date\ ...\}$$

and its dual gives the buyer's point of view:

$$!string\ ?int\ \oplus \{quit : end,\ ok :!string\ ?date\ ...\}$$

To see what multiparty session types can buy us, we'll follow [Honda et al.(2008)Honda, Yoshida, and Carbone] in generalising the problem to three parties.

Example 4. (Two-buyer protocol.) Seller S is selling an expensive book to whoever can afford it.

Buyers A and B wish to pool their money and collaborate in buying the book.

$$
\begin{array}{rcl}
S & = & start[3](a,b,s,ab). \\
 & & \quad s?title;\ a!quote;\ b!quote; \\
 & & \quad s \triangleright \{quit : \mathbf{0}, \\
 & & \qquad ok : s?address;\ b!date;\ ...\}
\end{array}
$$

$$
\begin{array}{rcl}
A & = & \overline{start}[2,3](a,b,s,ab). \\
 & & \quad s!title;\ a?quote;\ ab!(quote/2);\ ...
\end{array}
$$

$$
\begin{array}{rcl}
B & = & start[2](a,b,s,ab). \\
 & & \quad b?quote;\ ab?contrib; \\
 & & \quad \mathbf{if}\ quote - contrib \geq limit \\
 & & \qquad \mathbf{then}\ s \triangleleft quit \\
 & & \qquad \mathbf{else}\ s \triangleleft ok;\ s!address;\ b?date;\ ...
\end{array}
$$

Notice that we *could* use binary session types to describe the interactions between each pair of parties, but that in general this fails to capture some causality constraints. It also offers no way of delimiting the start or the end of a conversation. Instead, we write down a *global type* that directly and precisely captures the form of the whole conversation, including its initiation and termination.

5 The types

Definition 5. (Global types G, local types T, and port sorts U.)

$$
\begin{array}{rcll}
G & ::= & p_1 \to p_2 : k\langle U \rangle.G & \\
 & | & p_1 \to p_2 : k\{\ell_i : G_i\}_{i \in I} & \\
 & | & G, G & \text{(parallel composition)} \\
 & | & \mu \mathbf{t}.G & \\
 & | & \mathbf{t} & \\
 & | & end &
\end{array}
$$

$$
\begin{array}{rcl}
T & ::= & k!U;T \\
 & | & k?U;T \\
 & | & k \oplus \{\ell_i : T_i\} \\
 & | & k\&\{\ell_i : T_i\} \\
 & | & \mu \mathbf{t}.T \\
 & | & \mathbf{t} \\
 & | & end
\end{array}
$$

$$
U \quad ::= \quad string \mid int \mid ... \mid \langle G \rangle
$$

Example 6. A global type for the shared name *start* used in the two-buyer process given above:

$$
\begin{array}{rl}
start & : \ \langle \\
& \quad 1 \to 3 : s\langle string \rangle . \\
& \quad 3 \to 1 : a\langle int \rangle . \\
& \quad 3 \to 2 : b\langle int \rangle . \\
& \quad 1 \to 2 : ab\langle int \rangle . \\
& \quad 2 \to 3 : s\{quit : end, \\
& \qquad\qquad ok : \ 2 \to 3 : s\langle string \rangle . \\
& \qquad\qquad\quad 3 \to 2 : b\langle date \rangle . \cdots \} \\
& \quad \rangle
\end{array}
$$

6 Projection: From linear global types to local types

Definition 7. (Projection.) Let G be linear. (We'll see below what that entails.) Projecting G onto participant p is written $G \upharpoonright p$ and defined

$$
\begin{array}{rcll}
\_ \upharpoonright \_ : G \times \mathbb{N} & \to & T & \\
(p_1 \to p_2 : k\langle U \rangle . G) \upharpoonright p & = & k!U; (G \upharpoonright p) & \text{if } p = p_1 \\
 & & k?U; (G \upharpoonright p) & \text{if } p = p_2 \\
 & & G \upharpoonright p & \text{otherwise} \\
(p_1 \to p_2 : k\{\ell_i : G_i\}_{i \in I}) \upharpoonright p & = & k \oplus \{l_i : (G_i \upharpoonright p)\}_{i \in I} & \text{if } p = p_1 \\
 & & k\&\{l_i : (G_i \upharpoonright p)\}_{i \in I} & \text{if } p = p_2 \\
 & & G_1 \upharpoonright p & \text{if } p \notin \{p_1, p_2\} \text{and} \\
 & & & \quad \forall i, j \in I . (G_i \upharpoonright p) = (G_j \upharpoonright p) \\
(G_1, G_2) \upharpoonright p & = & G_1 \upharpoonright p & \text{if } p \in G_1 \text{and } p \notin G_2 \\
 & & G_2 \upharpoonright p & \text{if } p \in G_2 \text{and } p \notin G_1 \\
 & & end & \text{if } p \notin G_2 \text{and } p \notin G_1 \\
(\mu \mathbf{t} . G) \upharpoonright p & = & \mu \mathbf{t} . (G \upharpoonright p) & \\
\mathbf{t} \upharpoonright p & = & \mathbf{t} & \\
end \upharpoonright p & = & end &
\end{array}
$$

(We write $p \in G$ to mean that participant p is mentioned by some action in G.) Whereever a side condition fails to hold, projection is undefined. This is quite restrictive: note in particular it means that we cannot project a global type in which any participant is mentioned on both sides of a parallel-composition operator!

7 Pragmatics from projection

Where we can project a local type from a global type, we have something we can use to check the behaviour of a putative implementation of a participant in a session. The check can be made independently of the other participants, which gives us a nice way of using the system as a whole: a programmer can define some global session type G, project it into local types for each participant,

IO	Race	$A \to B : s,\ B \to C : s$	$s! \mid s?;s! \mid s?$
OI	Race	$A \to B : s,\ C \to A : s$	$s!;s? \mid s? \mid s!$
II	Race	$A \to B : s,\ C \to B : s$	$s! \mid s?;s? \mid s!$
OO/II	Safe	$A \to B : s,\ A \to B : s$	$s!;...;s! \mid s?;...;s?$
OO	Race/Safe	$A \to B : s,\ A \to C : s$	$s!;s! \mid s? \mid s?$

Table 1: Interesting causal orderings

and hand out the local types to different teams. Programmers from each team can independently and incrementally develop their implementations, checking them as they go against the local type they were given. (We'll see the rules for checking processes against local types below.) When they are done, the implementations can be combined, and the combination is known to be safe in the ways outlined above.

Example 8. (Projection onto buyer A.) If we project the global type given in example 6 onto buyer A, participant 1, we get

$$s!string;\ a?int;\ ab!int;\ end$$

Example 9. (projection onto buyer B.) If we project onto participant 2 instead, we get

$$b?int;\ ab?int;\ s \oplus \{quit : \mathbf{0},\ ok : s!string;\ b?date;\ \cdots\}$$

8 Linearity is a proxy for causality

In the definition, we required that G be "linear": projection isn't defined if G is not "linear". But what is linearity? It's a temporal ordering property of the way a global type uses channels, chosen to ensure race freedom.[2] It's based on a notion of dependencies between communication actions at a given channel.

Table 1 shows the interesting cases. Notice how they all involve pairs of actions taken (in parallel) on the same channel. The first column (IO, OI, etc.) describes the direction of causality: IO means the input action at the channel must be causally prior to the output action; II that an input action must precede another input action; etc.

Definition 10. (Prefix.) A piece of syntax n of the form "$p_1 \to p_2 : k$" in some G is called a *prefix from p_1 to p_2 at k in G*. We write $n \in G$ when G contains some prefix n.

Definition 11. (Prefix ordering.) We say $n_1 \prec n_2$ in some G if either

- $G = n_1\langle U \rangle.G'$ and $n_2 \in G'$, or
- $G = n_1\{\ell_i : G_i\}_{i \in I}$ and $\exists j \in I.n_2 \in G_j$, or
- $n_1 \prec n_2$ in some smaller G' contained within (but not carried by) G.

Note that this means that prefixes appearing in different parts[3] of a parallel composition are not ordered with respect to one another.

[2] Other, less restrictive approaches to race-freedom have been developed [Bettini et al.(2008)Bettini, Coppo, D'Antoni, De Luca, Dezani-Ciancaglini, a but not in the papers I'm covering here.

[3] I'm being careful not to call them *branches* of such a composition here, because that term is used to mean the selection/branching construct $\triangleleft/\triangleright$, and that construct *does* admit prefix ordering across it.

Definition 12. (Dependency relations.) Define $\prec_\phi$ with $\phi \in \{II, IO, OO\}$ to avoid the known races as seen in table 1.

$$\begin{aligned} n_1 \prec_{II} n_2 \quad & \text{if } n_1 \prec n_2 \text{and } n_1 = p_1 \to p : k_1 \text{and } n_2 = p_2 \to p : k_2 \\ n_1 \prec_{IO} n_2 \quad & \text{if } n_1 \prec n_2 \text{and } n_1 = p_1 \to p : k_1 \text{and } n_2 = p \to p_2 : k_2 \\ n_1 \prec_{OO} n_2 \quad & \text{if } n_1 \prec n_2 \text{and } n_1 = p \to p_1 : k \text{and } n_2 = p \to p_2 : k \end{aligned}$$

Definition 13. An *input dependency* from n_1 to n_2 is a chain

$$n_1 \prec_{\phi_1} \cdots \prec_{\phi_n} n_2$$

where the ϕ_i are either II or IO for all but the last, which has to be II.

Definition 14. An *output dependency* from n_1 to n_2 is a similar chain where all the ϕ_i are either OO or IO, and there is no special restriction on the last one.

Definition 15. (Linearity.) G is linear if, for all pairs of prefixes n_1 and n_2 at the same channel k that don't occur in different branches of G, either

- both input and output dependencies exist from n_1 to n_2; or
- both input and output dependencies exist from n_2 to n_1.

That is, we preserve enough causality to assure race-freedom along all the possible paths through G.

An interesting question arises around recursion: it turns out G is linear iff its first unfolding is linear.

Example 16. This global type is not linear:

$$\mu X.(A \to B : s.end,\ B \to A : t.X)$$

It looks linear in its 0th unfolding, but when we examine its first unfolding

$$\mu X.(A \to B : s.end,\ B \to A : t.(A \to B : s.end,\ B \to A : t.X))$$

we see that it's possible for there to be a race between two transmissions from participant A along channel s, after B has sent one transmission along channel t.

9 Incoherent global types: they don't make sense

Back to projections. There are some global types G that don't make sense when you project them out to local types, despite being linear and otherwise well-formed.

Definition 17. (Coherence.) A type G is coherent if it is linear and if $G \upharpoonright p$ is defined for every participant p in G.

Example 18. The following type is incoherent:

$$\begin{aligned} A \to B : k\{ \quad ok : \quad & C \to D : k'\langle bool \rangle, \\ quit : \quad & C \to D : k'\langle nat \rangle \} \end{aligned}$$

To see why, consider that C and D are totally independent of A and B, with no communication from either of A or B to either of them, and yet the type of the communication at k' depends on the selection made by A!

10 Typing programs

After all this trickiness about linearity, actually checking types for processes is quite easy.

Definition 19. (Sortings and typings.) Type judgements for processes are written

$$\Gamma \vdash P \triangleright \Delta$$

where

- Γ is a *sorting* mapping names bound in receive actions to the sort of the value received and mapping process variables to the sorts and local types required for their arguments, and
- Δ is a *typing* mapping each vector of session channels to a corresponding family of local types, one for each participant involved in the session. Roughly speaking, this is the collection of remaining actions that need to be performed at the session channels to discharge the global session type.

$$\begin{array}{rcl} \Gamma & ::= & \emptyset \mid \Gamma, u : U \mid \Gamma, X : \tilde{U}\tilde{T} \\ \Delta & ::= & \emptyset \mid \Delta, \tilde{s} : \{T@p\}_{p \in I} \end{array}$$

Unusually, typings Δ change during reduction! We'll see this in action below.

There is also a more traditional judgement form, $\Gamma \vdash e : U$ for the non-communicating expression fragment of the language.

I'll show a few of the more interesting type rules.

$$\frac{\Delta \text{ end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \quad \text{(inact)}$$

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathbf{if}\, e\, \mathbf{then}\, P\, \mathbf{else}\, Q \triangleright \Delta} \quad \text{(if)}$$

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright p)@p}{\Gamma \vdash a[p](\tilde{s}).P \triangleright \Delta} \quad \text{(macc)}$$

$$\frac{\Gamma \vdash e : U \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@p \quad k \in \tilde{s}}{\Gamma \vdash k!e; P \triangleright \Delta, \tilde{s} : k!U; T@p} \quad \text{(send)}$$

$$\frac{\Gamma, x : U \vdash P \triangleright \Delta, \tilde{s} : T@p \quad k \in \tilde{s}}{\Gamma \vdash k?x; P \triangleright \Delta, \tilde{s} : k?U; T@p} \quad \text{(recv)}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{s} : T_j@p \quad j \in I \quad k \in \tilde{s}}{\Gamma \vdash k \triangleleft \ell_j; P \triangleright \Delta, \tilde{s} : k \oplus \{\ell_i : T_i\}_{i \in I}@p} \quad \text{(sel)}$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta, \tilde{s} : T_i@p \quad \forall i \in I}{\Gamma \vdash k \triangleright \{\ell_i : P_i\}_{i \in I} \triangleright \Delta, \tilde{s} : k\&\{\ell_i : T_i\}_{i \in I}@p} \quad \text{(branch)}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta' \quad \Delta \asymp \Delta'}{\Gamma \vdash P | Q \triangleright \Delta \circ \Delta'} \quad \text{(conc)}$$

That last rule involves two operators on typings, $\circ$, which composes two partial typings, and $\asymp$, a predicate on typings deciding whether they are compatible or not.

First, $\circ$ is defined on collections of located local types, such as appear on the right in bindings within typings:

$$\{T_p@p\}_{p\in I} \circ \{T'_{p'}@p'\}_{p'\in J} = \{T_p@p\}_{p\in I} \cup \{T'_{p'}@p'\}_{p'\in J} \qquad \text{if } I \cap J = \emptyset$$

Next, $\circ$ is lifted to compose whole typings as follows:

$$\begin{array}{rcll} \Delta_1 \circ \Delta_2 & = & \{ \quad \Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s}) & \mid \quad \tilde{s} \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \\ & & \cup \quad \Delta_1 \backslash \mathsf{dom}(\Delta_2) & \\ & & \cup \quad \Delta_2 \backslash \mathsf{dom}(\Delta_1) & \end{array}$$

The compatibility of two typings is defined

$$\begin{array}{rcl} \Delta_1 \asymp \Delta_2 & = & \forall \tilde{s}_1 \in \mathsf{dom}(\Delta_1). \forall \tilde{s}_2 \in \mathsf{dom}(\Delta_2). \\ & & \tilde{s}_1 \cap \tilde{s}_2 \neq \emptyset \implies ((\tilde{s} = \tilde{s}_1 = \tilde{s}_2) \wedge (\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s}) \text{ defined})) \end{array}$$

In essence, two typings are compatible when every partially shared vector of session channels is in fact completely shared, and furthermore that the participants of such shared channel vectors are disjoint. This is another linearity constraint, keeping activity within a participant within a particular global type sequential.

11 Typing intermediate states

So far we've seen judgements for typing programs and program phrases as they would be written by a programmer. The full syntax for the language includes constructs that the programmer never writes explicitly, such as message queues $s : \tilde{h}$, so we must extend our type system to these to obtain a subject reduction theorem. The key idea is to use holes in our local types for queues; essentially, we introduce a kind of type context.

Example 20. After the process

$$s!3; s!true; \mathbf{0} \mid s : \emptyset \mid s?x; s?y; \mathbf{0}$$

reduces one step, we have the process

$$s!true; \mathbf{0} \mid s : 3 \mid s?x; s?y; \mathbf{0}$$

within which we type the queue, $s : 3$, as

$$s : \{s!int; \Box@p, \quad \Box@q\}$$

The types we assign to queues represent all the actions that have happened so far at that channel within a given participant. So for the example above, we see that participant p has so far output an *int* on channel s, while participant q has yet to take an action at channel s.

When the (conc) type rule is used above, two parallel processes have their typings composed with $\circ$. This tells us that when a queue process is composed with another process, $\circ$ should fill the hole in the queue process's typing, yielding a complete record of actions at each channel.

Definition 21. (Redefinition of typing.) Typings Δ are redefined as follows:

$$\begin{array}{rcl} \Delta & ::= & \emptyset \mid \Delta, \tilde{s} : \{H_p\}_{p\in I} \\ H & ::= & T \mid \mathcal{T} \\ T & ::= & \text{as before} \\ \mathcal{T} & ::= & \Box \mid k!U; \mathcal{T} \mid k \oplus \ell_i : \mathcal{T} \end{array}$$

There are two interesting things to note here:

1. $\mathcal{T}$ only includes *output* actions! This suggests that we have assigned "ownership" of the queue to the outputting participant, for the purposes of type checking. Once the composition of the queue with the remainder of the necessary output actions has been judged, a complete (hole-free) typing is obtained, which is consistent with the typing for the process before any actions were taken at the channel corresponding to the queue.

2. The production $k \oplus \ell_i : \mathcal{T}$ only includes a single label. Honda et al. introduce a subtyping relation to let the selection represented by a label sitting in a queue be made compatible with the full menu of selections available to a participant.

   In [Honda et al.(2008)Honda, Yoshida, and Carbone], the authors note that these types for queues in a sense permit "rolling back" the state of a participant to before anything happened on the queue. The subtyping for selection is needed to widen the possibilities to say, effectively, "although I chose this option, I *could have* chosen from any of these options".

Type judgements are changed to include the set of message queues "belonging" to a given branch of the proof tree: $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. The (conc) rule partitions queues across parallel composition similarly to the way it partitions participants' behaviours.

$$\frac{\Gamma \vdash v : U \quad \Gamma \vdash k : \tilde{h} \triangleright_k \Delta, \tilde{s} : (\{\mathcal{T}@p\} \cup R) \quad R = \{H_q@q\}_{q \in I}}{\Gamma \vdash k : \tilde{h} \cdot v \triangleright_k \Delta, \tilde{s} : (\{\mathcal{T}[k!U;\Box]@p\} \ \cup \ R)} \qquad \text{(qval)}$$

$$\frac{\Delta \text{ end only}}{\Gamma \vdash k : \emptyset \triangleright_k \Delta \circ (\tilde{s} : \{\Box@p\})} \qquad \text{(qnil)}$$

12 Typings reduce, analogously to reduction of processes

Above, I mentioned that types change at runtime. This makes sense, in that a typing represents the actions remaining within a session. Type reduction, then, is defined as follows:

$$\begin{aligned} k!U;H@p,\ k?U;T@q &\rightarrow H@p,\ T@q \\ k \oplus \{\ell : H, ...\}@p,\ k\&\{\ell : T, ...\}@q &\rightarrow H@p,\ T@q \end{aligned}$$

It is then lifted into typings $\Delta \rightarrow \Delta'$ and from there into entire graphs $G \rightarrow G'$ via projection into local types.

We now have enough to be able to state a subject reduction theorem.

Theorem 22. *(Subject congruence and reduction.)*

1. $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *and* $P \equiv P'$ *imply* $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta$.
2. $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *such that* Δ *is coherent*[4] *and* $P \rightarrow P'$ *imply* $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ *where either* $\Delta = \Delta'$ *or* $\Delta \rightarrow \Delta'$.
3. $\Gamma \vdash P \triangleright_{\emptyset} \emptyset$ *and* $P \rightarrow P'$ *imply* $\Gamma \vdash P' \triangleright_{\emptyset} \emptyset$.

From here, the paper goes on to state communication safety, session fidelity, and progress theorems in full detail. I'll draw a veil over these theorems, however, and we'll move on to the interesting results from the second paper, [Mostrous et al.(2009)Mostrous, Yoshida, and Honda].

[4] I haven't defined that here: essentially, if all the $\Delta(\tilde{s})$ in Δ when recomposed into a global type make a coherent such type.

13 Pragmatic benefits of asynchronous subtyping

[Mostrous et al.(2009)Mostrous, Yoshida, and Honda] introduces a new kind of subtyping, and then uses it to show type-safe optimisations of processes as well as to infer principal global types for collections of processes. This has two important pragmatic benefits:

1. It can make distributed systems written in this style more efficient, by permitting the system to automatically expose more asynchrony, thus reducing latency; and
2. It enables an entirely new style of working with the system: without type inference, we had to proceed strictly from designing a global type G, projecting it to a collection of local types $T@p$, and then checking our implementations P against each local type.

   With type inference, however, we can now incrementally develop our implementations P, infer local types $T@p$, and compose them into a global type G which by use of the asynchronous subtyping system is guaranteed to be a principal type for our processes!

14 When is it safe to reorder actions?

Say I have a process with local type

$$k?U; k'!U'; T$$

So long as the value sent down k' is not causally connected to the value received from k, it's safe to reorder this to be

$$k'!U'; k?U; T$$

by pushing the input action under the output action.

The only unsafe cases are pushing output or selection actions under input or branching actions, because to do so could cause a deadlock.

Example 23. If we reorder the process $s!; r? \mid r!; s?$ by pushing the output actions under the input actions, we are left with $r?; s! \mid s?; r!$, which is a deadlocked process.

There's a nice matrix showing when it's safe to reorder actions on distinct channels:

Push $\downarrow$ under $\rightarrow$	Output	Input	Selection	Branching
Output	✓	×	.	×
Input	✓	✓	.	.
Selection	.	×	.	×
Branching	.	.	.	.

Where ✓ means a simple reordering is safe, × means reordering is not safe, and . means that reordering is safe so long as the reordered actions are applied equally in all the possible combinations of continuations.

Example 24. It's safe to push the input under the selection in $k?U; k' \oplus \{\ell_i : T_i\}$ to yield $k' \oplus \{\ell_i : (k?U; T_i)\}$, because the input has been applied to all the branches, but it would be incorrect to push it under just one of the labels.

This reordering yields a relation $T \ll T'$, read T is an *action-asynchronous subtype* of T'. That is, T is more asynchronous than T', and T can be obtained by reordering actions in T' as per the matrix above.[5]

[5]Strictly speaking, as per the formal rules in the paper. The matrix is just a visual summary of the rules.

The relation $T \ll T'$ is extended to a coinductive subtyping relation $T \leq_c T'$ ("T is an *asynchronous subtype* of T'") by unfolding of recursion in the types and application of a method similar to the logical relations we've seen in class. Finally, $\leq$, an algorithmic equivalent of $\leq_c$, is given and then used in a subsumption rule extending the typing rules sketched above.

15 Limitations of the system

Linearity. The fact that global types G must be linear in order to be projected is a strong restriction. Certain kinds of race are desirable. Consider the useful and interesting process describing a sequence of message transmissions from A to B, with acknowledgements from B to A able to be both delayed and also sent in an order different from the order of receipt of the corresponding messages:

$$\langle \mu \mathbf{t}.\ 1 \to 2 : data\langle message \rangle.\ ((2 \to 1 : ack\langle message \rangle.end)\ ,\ \mathbf{t}) \rangle$$

This is not linear after one unfolding: two $2 \to 1 : ack$ communications appear in parallel.

Even if we give up on letting acknowledgements be reordered, wanting them only to be delayed a while, during which delay further messages can be delivered, we run into trouble. I can't think of a way to write down a process that simply delays acknowledgements without also reordering them.

Parallel composition. It's unclear how useful the parallel composition operator is in a global type, since there are such strong restrictions on projection through a parallel composition. Could it be useful for larger "mashup"-style services, where after a transaction is begun, two independent sub-transactions proceed in parallel without communicating with each other? The examples given in the papers I've seen are too small to tell.

One way of avoiding the restrictions might be to exploit the fact that processes can be running in several different sessions (each independently typed) simultaneously, arbitrarily interleaving communications typed by each session.[6] So complex sessions requiring untypable behaviour might be able to be split into smaller sessions for which typing does go through; by doing this, however, one forfeits some of the advantages of static types.

Appendix: The papers

These notes were by-and-large based on the following two papers.

Multiparty Asynchronous Session Types; POPL 2008

Kohei Honda, Nobuko Yoshida, Marco Carbone.

Abstract

Communication is becoming one of the central elements in software development. As a potential typed foundation for structured communication-based programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focussing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centred applications. Presented as a typed calculus

[6] Whether this is actually a fact is unclear, though it does look *possible* to me, from my current limited understanding of the system.

for mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types retain a friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual peers. The fundamental properties of the session type discipline such as communication safety and progress are established for general n-party asynchronous interactions.

Outline

Gives an example of a 2-party session type. Outlines desirable properties session types enforce. Motivates the generalisation to an n-party setting. Syntax & semantics of processes. Detailed examples. Syntax and meaning of types. Safety principle for global types. Projection from global to local types; type checking. Safety & progress properties of processes.

Contributions

Generalisation of session types from 2-party to n-party. Causality analysis of interaction, leading to linearity analysis and so safety and progress properties. Projection of global types onto individual session participants.

Global Principal Typing in Partially Commutative Asynchronous Sessions; ESOP 2009

Dimitris Mostrous, Nobuko Yoshida, Kohei Honda. Can be seen as an extension of the work begun in the previous paper; also incorporates some of the results in [Bettini et al.(2008)Bettini, Coppo, D'Antoni, De Luca, Dezani-

Abstract

We generalise a theory of multiparty session types for the π-calculus through asynchronous communication subtyping, which allows partial commutativity of actions with maximal flexibility and safe optimisation in message choreography. A sound and complete algorithm for the subtyping relation, which can calculate conformance of optimised end-point processes to an agreed global specification, is presented. As a complementing result, we show a type inference algorithm for deriving the principal global specification from end-point processes which is minimal with respect to subtyping. The resulting theory allows a programmer to choose between a top-down and a bottom-up style of communication programming, ensuring the same desirable properties of typable processes.

Outline

Recaps the top-down approach from the previous paper. Introduces a bottom-up approach, made possible by the principal type reconstruction algorithm given later in the paper. Describes optimisations possible from their analysis of asynchronous communication subtyping. Recaps multiparty asynchronous session types, as per the previous paper but slightly simplified. Defines asynchronous communication subtyping, first by exploring when it is safe to permute action prefixes, then by defining a declarative subtyping relation, and finally by defining algorithmic subtyping rules and proving them equivalent to the declarative rules. Extends the local type system from the previous paper to

incorporate asynchronous communication subtyping. Presents an alternative graph-based perspective on the global types developed in the previous paper, arguing that it is a more useful approach to working with them. Develops a method of discovering a principal global typing from local processes. Gives an example of the system as applied to a double-buffering stream processing algorithm.

Contributions

A new, bottom-up, approach to working with multiparty session types. A subtyping system that permits not only optimisation of distributed systems for increased asynchrony, but that also permits global type reconstruction from process terms.

References

[Bettini et al.(2008)Bettini, Coppo, D'Antoni, De Luca, Dezani-Ciancaglini, and Yoshida] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008. URL `http://www.di.unito.it/ dezani/papers/bcdddyshort.pdf`. Long version at http://www.di.unito.it/ dezani/papers/cdy12.pdf.

[Fournet and Gonthier(2000)] Cédric Fournet and Georges Gonthier. The Join Calculus: a Language for Distributed Mobile Programming, 2000.

[Honda et al.(1998)Honda, Vasconcelos, and Kubo] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. *Lecture Notes in Computer Science*, 1381:122–138, 1998. URL `http://www.springerlink.com/index/fakx696bw3vx5kgr.pdf`.

[Honda et al.(2008)Honda, Yoshida, and Carbone] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, volume 43, pages 273–284, San Francisco, California, January 2008. ISBN 9781595936899. doi: 10.1145/1328438.1328472. URL `http://dl.acm.org/citation.cfm?id=1328438.1328472`.

[Milner(1991)] Robin Milner. The Polyadic π-Calculus: a Tutorial. 1991.

[Mostrous et al.(2009)Mostrous, Yoshida, and Honda] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP '09*, pages 316–332, March 2009.

[Pierce and Sangiorgi(1996)] Benjamin C. Pierce and Davide Sangiorgi. Typing and Subtyping for Mobile Processes. *MATHEMATICAL STRUCTURES IN COMPUTER SCIENCE*, 6:376 – 385, 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.415`.