A linear type system for pi calculus

joint work with Vasco T. Vasconcelos

Marco Giunti
INRIA & LIX, École Polytechnique Palaiseau

Behavioral Types, April 19 2010, Lisboa
Session Types

- Describe a protocol between a service provider and a client
- Introduced for the pi calculus and now embedded also in other paradigms based on message passing
  - functional programming
  - object oriented programming
- Idea: allowing typing of channels by using structured sequences of types as output, output, input, ..
  
  !Integer . !Boolean . ? Boolean . end
Session types in the pi calculus

• In [HVK Esop’98] a typing discipline for structured programming is introduced for a dialect of pi calculus.

• **Session channels** are used to abstract binary sessions and are distinguished from standard pi calculus channels or names.

• Session initiation arises on names.

• Fidelity of sessions is guaranteed by a typing system enforcing a session channel to be used at most by two threads with opposite capabilities (e.g. input/output).
Discussion

• In the original system and recent works session delegation is restricted to bound output

\[
\overline{x\langle k\rangle}.P \parallel x(k).Q \rightarrow P \parallel Q
\]

• Communication mechanism of the pi calculus breaks subject reduction

• Decoration of channel end-points is the de-facto workaround [GH Acta’05]

\[
\overline{x^+\langle y^p\rangle}.P \parallel x^-(z).Q \rightarrow P \parallel Q[y^p/z]
\]

• Distinction between names and session channels of [HVK98] leads to duplicate typing rules
What we have done

- Remove distinction among session channels and names
- Do not use polarities or double binders
- That is: we use standard pi calculus
- Annotate session types with qualifiers
  - \texttt{lin} for linear use
  - \texttt{un} for unrestricted use
- Introduce a type construct that describes the two ends of a same channel
Types

- **Types** $T$
  - $S$ for *end point type* describing one channel end
  - $(S, S)$ for *channel type* describing both channel ends

- **End point types** $S$ are
  - $\text{lin}_p$ linear channel used exactly once
  - $\text{un}_p$ channel is used zero or more times
  - $\mu a.S$ and $a$ for recursive end point types

- **Session types** $p$ are
  - $?T.S$: waits for value of type $T$ then continues as $S$
  - $!T.S$: sends a value of type $T$ then continues as $S$
  - end: no further interactions are possible
Example: event scheduling

1. Create poll
   - provide the title for the meeting
   - provide a provisional date

2. Invite participants
   - Pi calculus: send request to create poll / receive poll channel
   \[
   \overline{\text{poll}}\langle y \rangle.y(p).(p\langle \text{Workshop} \rangle.p\langle 19\text{April} \rangle.(\overline{z_1}(p) \mid \cdots \mid \overline{z_n}(p)))
   \]
   - Challenge: concurrent distribution of the poll channel
Session type for the poll

• Poll channel used first in linear mode then in unrestricted mode

• Steps:
  1. Send a title for the poll (linear mode)
  2. Send a date for the poll (linear mode)
  3. Distribute the poll (unrestricted mode)

\[
y(p). (p\langle\text{Workshop}\rangle.p\langle19\text{April}\rangle. (z_1(p) \mid \cdots \mid z_n(p) ))
\]

• End point session type for channel \( p \) is

\[
\text{lin } !\text{string}. \text{lin } !\text{date}. * S \quad \text{where} \quad * S = \text{un } !\text{date}. * S
\]

• Recursive unrestricted type \( S \) allows distribution of poll channel
Type for the scheduling service

- Service: instantiation generates poll

\[
Service = !poll(w). (\nu p : (S_1, S_2)) \ (\overline{w}(p). p(t). p(d). !p(d))
\]

\[
S_1 = \text{lin } ?\text{string. lin } ?\text{date. } *\text{un } ?\text{date}
\]

\[
S_2 = \text{lin } !\text{string. lin } !\text{date. } *\text{un } !\text{date}
\]

- Poll channel is **split**:
  1. One channel end sent to the **invoker**
  2. The other channel end used in the **continuation**
Context splitting

- Type system $\Gamma \vdash P$ based on context splitting $\Gamma_1 \cdot \Gamma_2$

- Unrestricted types are copied into both contexts

- Linear types are placed in one of the two resulting contexts

\[
\begin{align*}
\Gamma_1, p : S_2 & \vdash p : S_2 \\
\Gamma_2, w : \text{end}, p : S_1 & \vdash p(t).p(d).!p(d) \\
\Gamma & = \Gamma_1 \cdot \Gamma_2 \\
\Gamma, w : \text{lin} !S_2.\text{end}, p : (S_1, S_2) & \vdash \overline{w} \langle p \rangle .p(t).p(d).!p(d)
\end{align*}
\]
Subject reduction

• $\Gamma$ balanced, $\Gamma \vdash P, P \rightarrow P'$ imply $\Gamma' \vdash P'$ with $\Gamma'$ balanced

• Interesting case: $(q ?T.S_1, q ?T.S_2)$ is balanced if both $T$ and $(S_1, S_2)$ are balanced

• Purpose of balancing is to preserve soundness of exchange

\[
\Gamma = x : (\text{lin } ?(\ast !\text{bool}).\text{un end}, \text{lin } !(\text{un end}).\text{un end}), y : \text{un end}
\]
\[
\Gamma \vdash x(z) . \bar{z} \langle \text{true} \rangle | \bar{x} \langle y \rangle
\]
\[
x(z) . \bar{z} \langle \text{true} \rangle | \bar{x} \langle y \rangle \rightarrow \bar{y} \langle \text{true} \rangle
\]
\[
x : (\text{un end, un end}), y : \text{un end} \not\vdash \bar{y} \langle \text{true} \rangle
\]
SR at work

- Receiving of a session already known

\[ \overline{x}\langle v \rangle \mid x(y).\overline{v}\langle \text{true} \rangle .y(z) \rightarrow \overline{v}\langle \text{true} \rangle .v(z) \]

- Typing the redex

\[
\begin{align*}
\Gamma & : (\text{un end}, \text{lin } ?\text{bool.un end}) \vdash v(z) \\
\Gamma & : (\text{lin } !\text{bool.un end}, \text{lin } ?\text{bool.un end}) \vdash \overline{v}\langle \text{true} \rangle .v(z)
\end{align*}
\]
Algorithm

- Type system $\vdash$ cannot be implemented directly

- Main difficulty is split operation

- We avoid split by
  1. passing entire context for the judgement
  2. mark linear types consumed in the derivation as *unusable*
Type checking

- Algorithm relies on several patterns of checking function
  
  ```
  fun check(g : context, p : process) : context
  ```

- Context in input is balanced
  1. patterns are non ambiguous
  2. no backtracking is needed

- Context in output has **void** marks in place of consumed types

- Top-level call accepts process if check returns unrestricted context
  
  ```
  fun typeCheck(g : context, p : process) : bool
  ```
Checking the service

- Poll delegation: type for delegation channel $T = \text{lin} !S_2 . \text{un end}$
  
  \[ \text{check}(\Gamma, w : T, p : (S_1, S_2) , \overline{w}(p) . P) = \]
  
  \[ \text{let val } d = \text{check}(\Gamma, w : \text{un end}, p : (S_1, \circ), P) \]
  
  \[ \text{in if } d = d', w : M \text{ and } M = \circ, \text{un } p \text{ then } d', w : \circ \]

- Call for the continuation by setting delegated end point for the poll to **void** (noted $\circ$)

- Linear use of channel must be consumed within the continuation (condition $M = \circ, \text{un } p$)

- Returned context obtained by setting to void the unrestricted type for the channel
Checking the continuation

- Linear receiving of the date: \( S_1 = \text{lin } ? \text{string} \cdot \text{lin } ? \text{date} \cdot * \text{un } ? \text{date} \)

\[
\text{check}(\Gamma, p : (S_1, N), p(t).P) =
\]

\[
\begin{array}{c}
\text{let val } d = \text{check}(\Gamma, p : \text{lin } ? \text{date} \cdot * \text{un } ? \text{date}, t : \text{string}, P) \\
\text{in if } d = d', p : M \text{ and } M = \circ, \text{un } p \text{ then } d', p : (\circ, N)
\end{array}
\]

- Checking of the continuation invoked by passing one channel end

- Linear use of channel must be consumed within the continuation (condition \( M = \text{un } p, \circ \))

- Returned context re-builds channel type by setting used channel end to void
Checking the scheduling protocol

• Protocol described by concurrent execution of

\[ Service = !poll(w). (\nu p) (\overline{w} p) . p(t) . p(d) . !p(d) ) \]

\[ Invoker = poll(y) . y(p) . (p⟨Workshop⟩ . p⟨19April⟩ . (\overline{z}_1 p | .. | \overline{z}_n p)) \]

• Type checking

\[
\text{check}(\Gamma, Service | Invoker) = \\
\quad \text{check}( Invoker, \text{check}(\Gamma, Service) )
\]

• Preservation of structural congruence

\[
\text{check}(\Gamma, Invoker | Service) = \text{check}(\Gamma, Service | Invoker)
\]
Algoritmic soundness

- The algorithm is sound
  - typeCheck(Γ, P) implies Γ ⊢ P

- Completeness missing since ⊢ permits to infer
  - Γ, x : (lin ?T.S₁, lin !T.S₂) ⊢ ⟨v⟩P
  - Γ, x : (lin ?T.S₁, lin !T.S₂) ⊢ ⟨v⟩P
  - Γ, x : (lin ?T.S₁, lin !T.S₂) ⊢ x⟨v⟩.C[x(y).P]

- Claim: processes in these judgements are deadlocked
Towards algorithmic completeness

• Proof transformation: $\Gamma_1 \vdash P_1$ transformed in $\Gamma_2 \vdash P_2$

• Construction: $\Gamma, x : (\text{lin } ?T.S_1, \text{lin } !T.S_2) \vdash \overline{x}\langle v \rangle.Q$ substituted in the derivation tree for $\Gamma_1 \vdash P_1$ with $\emptyset \vdash 0$

• Typed equivalence: $\Gamma_1 \triangleright P_1$ and $\Gamma_2 \triangleright P_2$ have same behavior
  - $\Gamma \triangleright P$ is typed configuration such that $\Delta \vdash P$ and $\Gamma \cdot \Delta$ defined
  - $\Gamma$ is less informative typed observer allowing moves of $P$

• Semantic completeness: typeCheck ($\Gamma_2, P_2$)
Conclusions

• We introduced type system $\vdash$ based on construct that describes the two ends of the same channel
  
  - An end point is described by session type qualified as linear or unrestricted
  
  - Linear types evolve to unrestricted types

• We assessed expressiveness by defining type-preserving encoding of
  
  1. linear lambda calculus [Walker&05]
  2. linear pi calculus [KPT TOPLAS’99]
  3. pi calculus with polarities [GH Acta’05]
Ongoing and future work

• We implemented rules ⊢ in type checking algorithm

• (Semantic) completeness in progress

• Still there are interesting processes that are not typable by ⊢

\[ !x(y).(\nu a)(\bar{y}a\cdot a(title)\cdot a(date)\cdot (!a(date) \mid \bar{a}(22\text{March})) \]

• Both capabilities needed in continuation for receive and send date

• Sub typing à la Pierce&Sangiorgi would fix this