comète

A type checking algorithm for qualified session types

Marco Giunti

INRIA & LIX, École Polytechnique Palaiseau

WWV June 9 2011, Reykjavik

Structured-oriented programming SOP

- Session types are a static analysis technique for service oriented protocols

- Allow for analyzing the message-passing interaction among a service provider and a client

- Introduced for the pi calculus and now embedded also in other paradigms:
  - functional programming
  - object oriented programming

- Idea: allowing typing of communication channels by using polymorphic *sequences* of types as:

  output Integer . output Boolean . input Boolean . end

Qualified session types

- Session types core: input, output and termination

$?T.S$	input
$!T.S$	output
end	termination

- Qualifiers

lin$?T.S$	linear input
lin$!T.S$	linear output
$S_1 =$ un$?T.S_1$	unrestricted recursive input
$S_2 =$ un$!T.S_2$	unrestricted recursive output
un end	termination

SOP in the pi calculus

- Example: event scheduling (e.g. Doodle)
- Two-steps service protocol for scheduling a meeting (client side)
  1. Ask to create a poll
     - provide a title for the meeting
     - provide a provisional date
  2. Invite participants
- Implementation: send request to create poll / receive poll channel

$$\overline{\textit{poll}}\langle y\rangle.y(p).(\overline{p}\langle \text{Workshop}\rangle.\overline{p}\langle \text{9June}\rangle.(\overline{z_1}\langle p\rangle \mid \cdots \mid \overline{z_n}\langle p\rangle))$$

- Challenge: concurrent distribution of the poll channel

Session type for the poll

- Poll channel used first in linear mode then in unrestricted mode
  1. Send a title for the poll (linear mode)
  2. Send a date for the poll (linear mode)
  3. Distribute the poll (unrestricted mode)

$$\overline{p}\langle \text{Workshop} \rangle.\overline{p}\langle \text{9June} \rangle.(\overline{z_1}\langle p \rangle \mid \cdots \mid \overline{z_n}\langle p \rangle)$$

- End point session type for channel p is

$$\mathsf{lin}\,!\mathsf{string}.\mathsf{lin}\,!\mathsf{date}.{*}\mathsf{un}\,!\mathsf{date}$$

- Recursive type ${*}\mathsf{un}\,!\mathsf{date}$ allows for distribution of poll channel

Concurrent session types

- Service: instantiation generates poll

$$\textit{Service} = !\textit{poll}(w).(\nu p : T)\overline{w}\langle p \rangle.p(\textit{title}).p(\textit{date}).!p(\textit{date})$$

- One channel end sent to the invoker

$$S_1 = \mathsf{lin}\,!\mathsf{string}.\mathsf{lin}\,!\mathsf{date}.*\mathsf{un}\,!\mathsf{date}$$

- The other channel end used in the continuation

$$S_2 = \mathsf{lin}\,?\mathsf{string}.\mathsf{lin}\,?\mathsf{date}.*\mathsf{un}\,?\mathsf{date}$$

- Full type for p describes *concurrent* behavior of two channel ends

$$T = (S_1, S_2)$$

This talk

- We present a type checking algorithm for qualified session types of the form
$$T = (S_1, S_2)$$

- The algorithm can be seen as an implementation of type system $\vdash$ of [G&V@Concur'10]

- Soundness proved by resorting to $\vdash$

- We discuss ongoing work on semantic completeness

Algorithmic type checking

- Well-known idea: in typing $P \mid Q$ remove linear identifiers used by P before type check Q (e.g. [Gay&Hole'05])

- Our approach: we reason at the type level and **forbid** ($\circ$) use of (parts of) types that have been:

   1. delegated

   2. consumed

$$(\sim ML) \quad \text{fun typeVar}(\Gamma, x\colon (\mathsf{lin}?T_1.S_1, \mathsf{lin}!T_2.S_2)\ ,\ x\colon \mathsf{lin}?T_1.S_1) = \Gamma, x : (\circ, \mathsf{lin}!T_2.S_2)$$

ML patterns

- Typings for processes are patterns of function:

$$\mathsf{fun\ check}(\mathsf{g} : \mathsf{context}, \mathsf{p} : \mathsf{process}) : \mathsf{context}$$

- Patterns matching deterministic for *safe* types (no backtracking)

  1. dual unrestricted channel ends: $(*\mathsf{un}?T, *\mathsf{un}!T)$ and T safe

  2. dual linear ends: $(\mathsf{lin}?T.S_1, \mathsf{lin}!T.S_2)$ and T and (S_1, S_2) safe

- E.g.: typing an input process in linear mode

$$\mathsf{check}(\Gamma, x\colon (\mathsf{lin}?T.S_1, \mathsf{lin}!T.S_2)\ ,\ x(y).P) =$$

$$\mathsf{let\ val\ d}\ = \mathsf{check}(\Gamma, x\colon S_1, y : T\ ,\ P)\ \mathsf{in}\ \ldots$$

Motivation of the design

- Algorithm works only for safe types = generalization *balancing*
- As usual, subject reduction only for balanced contexts
- The very reason is to preserve the soundness of the exchanges

$$P = x(y).\mathsf{if}\ y\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0} \mid (\nu z : \mathsf{end})\overline{x}\langle z\rangle.\mathbf{0}$$

$$x : \big(\mathsf{lin}?\mathsf{bool}.\mathsf{end}, \mathsf{lin}!\mathsf{end}.\mathsf{end}\big) \vdash P$$

$$P \longrightarrow \big(\nu z : \mathsf{end}\big)\mathsf{if}\ z\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$$

$$\not\vdash \big(\nu z : \mathsf{end}\big)\mathsf{if}\ z\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$$

- Soundness: algorithm rejects non balanced types

Type checking algorithm

- The top level call accepts the process if:

1. The environment in input is safe
2. An environment is given in output (no patterns exception)
3. The domain of the environment in output contains only *consumed* types of the form $\circ$, $\mathsf{un}\, p$, $(\mathsf{un} p_1, \mathsf{un} p_2)$, $(\mathsf{un} p, \circ)$, $(\circ, \circ)$

$$
\begin{array}{l}
\mathsf{fun\ typeCheck}(\Gamma : \mathsf{context}, P : \mathsf{process}) : \mathsf{bool} = \\
\quad \mathsf{if\ safe}(\Gamma)\ \mathsf{then} \\
\qquad \mathsf{let\ val}\ \Delta = \mathsf{check}(\Gamma, P)\ \mathsf{in} \\
\qquad\quad \mathsf{if}\ \mathsf{consumed}(\Delta)\ \mathsf{then\ true}
\end{array}
$$

A run

- Protocol described by concurrent execution of

$$\textit{Service} = !\textit{poll}(w).(\nu p : T)\ (\overline{w}\langle p\rangle.p(\textit{title}).p(\textit{date}).!p(\textit{date}))$$

$$\textit{Invoker} = \overline{\textit{poll}}\langle y\rangle.y(p).(\overline{p}\langle \mathsf{Workshop}\rangle.\overline{p}\langle 9\mathsf{June}\rangle.(\overline{z_1}\langle p\rangle \mid .. \mid \overline{z_n}\langle p\rangle))$$

$$S_1 = \mathsf{lin}\,!\mathsf{string}.\mathsf{lin}\,!\mathsf{date}.*\mathsf{un}\,!\mathsf{date}$$

$$S_2 = \mathsf{lin}\,?\mathsf{string}.\mathsf{lin}\,?\mathsf{date}.*\mathsf{un}\,?\mathsf{date}$$

$$T = (S_1, S_2)$$

$$T_w = \mathsf{lin}!S_1.\mathsf{un\ end}$$

- Type checking succeeds

$$\mathsf{typeCheck}(\Gamma, \textit{poll} : (*\mathsf{un}\,?T_w, *\mathsf{un}\,!T_w)\ ,\ \textit{Service} \mid \textit{Invoker})$$

Checking the service continuation

- Replicated input spawns a thread for the poll

$$\textit{Service} = !C$$

$$C = \textit{poll}(w).(\nu p : T)\overline{w}\langle p\rangle.p(\textit{title}).p(\textit{date}).!p(\textit{date})$$

- Call requires environment in output = environment in input

- Intuition: only linear types change!

$$\begin{array}{l} \mathsf{check}(\Gamma, \textit{Service}) = \\ \quad \mathsf{let\ val}\ \Delta = \mathsf{check}(\Gamma, C)\mathsf{in} \\ \quad\quad \mathsf{if}\ ({\color{red}\Delta = \Gamma})\ \ \mathsf{then}\ \Delta \end{array}$$

Checking unrestricted input

- Service instantiation generates poll

$$C = \textit{poll}(w).(\nu p : (S_1, S_2))\, \overline{w}\langle p\rangle.p(\textit{title}).p(\textit{date}).!p(\textit{date})$$

$$C' = (\nu p : (S_1, S_2))\, \overline{w}\langle p\rangle.p(\textit{title}).p(\textit{date}).!p(\textit{date})$$

$$T_w = \mathsf{lin}!S_1.\mathsf{un\ end}$$

- Call: type of channel does not change, bound variable added

- Return: checks types for the bound variable to be consumed

$$\begin{array}{l} \mathsf{check}(\Gamma, \textit{poll} : (*\mathsf{un}\,?T_w, *\mathsf{un}\,!T_w)\ , C) = \\ \quad \mathsf{let\ val}\ \Delta = \mathsf{check}(\Gamma, \textit{poll} : (*\mathsf{un}\,?T_w, *\mathsf{un}\,!T_w), w : T_w\ ,\ C')\ \mathsf{in} \\ \qquad \mathsf{if}\ (\Delta = \Delta', w : S)\ \mathsf{and}\ (S = \circ\ \mathsf{or}\ S = \mathsf{un}\, p)\ \mathsf{then}\ \Delta' \end{array}$$

Checking restriction

- A poll with safe channel type is generated

$$C' = (\nu p : (S_1, S_2))\overline{w}\langle p\rangle.Q$$

$$\Omega = \Gamma, \textit{poll} : (*\mathsf{un}\,?T_w, *\mathsf{un}\,!T_w), w : T_w$$

- Call: add bound variable given that the type is safe
- Return: checks type for bound variable to be consumed

$\mathsf{check}(\Omega\,, C') =$

$\;\mathsf{if}\ \mathsf{safe}((S_1, S_2))\ \mathsf{then}$

$\quad\mathsf{let\ val}\ \Delta = \mathsf{check}(\Omega, p : (S_1, S_2)\ ,\ \overline{w}\langle p\rangle.Q)\ \mathsf{in}$

$\qquad\mathsf{if}\ (\Delta = \Delta', p : (S', S''))$

$\qquad\quad\mathsf{and}\ (S' = \circ\ \mathsf{or}\ S' = \mathsf{un}\,p)$

$\qquad\quad\mathsf{and}\ (S'' = \circ\ \mathsf{or}\ S'' = \mathsf{un}\,p)$

$\qquad\mathsf{then}\ \Delta'$

Delegation of a linear session

- Poll write capability S_1 delegated over w of type $T_w = \text{lin}\,!S_1.\text{un end}$

$$\overline{w}\langle p \rangle.Q$$

- Call for the continuation
  1. session type for the channel unrolled
  2. delegated end point S_1 set to $\circ$ by calling fun checkVar
- Return
  1. checks type for channel to be consumed
  2. returns context with type for channel set to $\circ$

$$\text{check}(\Omega_1, w : T_w, p : (S_1, S_2)\ ,\ \overline{w}\langle p \rangle.Q) = \text{ let... in}$$

$$\text{let val } \Delta = \text{check}(\Omega_1, w : \text{un end}, p : (\circ, S_2)\ ,\ Q) \text{ in}$$

$$\text{if } \Delta = \Delta', w : S \text{ and } S = \circ \text{ or } S = \text{un}\, p \text{ then } \Delta', w : \circ$$

Checking parallel processes

- Concurrent delegation of poll channel to participants

$$P = \overline{z_1}\langle p\rangle \mid \overline{z_2}\langle p\rangle \mid \cdots \mid \overline{z_n}\langle p\rangle$$

$$S = {*}\mathsf{un}\,!\mathsf{date}$$

$$\Gamma = \Gamma_1, z_1 : \mathsf{lin}!S.\mathsf{end}, \cdots, z_n : \mathsf{lin}!S.\mathsf{end}, p : (\circ, S)$$

- Parallel processes typed compositionally

$$\begin{aligned}&\mathsf{check}(\Gamma\ ,\ P) = \\ &\quad \mathsf{let\ val}\ \Delta = \mathsf{check}(\Gamma\ ,\ \overline{z_1}\langle p\rangle)\ \mathsf{in} \\ &\qquad \mathsf{check}(\Delta\ ,\ \overline{z_2}\langle p\rangle \mid \cdots \mid \overline{z_n}\langle p\rangle)\end{aligned}$$

- In each call, the type of p in the input environment is $(\circ, S)$

Exchanging compositions' order

- Type checking the service protocol

$$\text{check}(\Gamma\,,\textit{Service}\mid\textit{Invoker}) = \text{check}\left(\text{check}\left(\Gamma\,,\ \textit{Service}\right)\,,\ \textit{Invoker}\right)$$

- Preservation of structural congruence!

$$\text{check}(\Gamma\,,\ \textit{Invoker}\mid\textit{Service}) = \text{check}(\Gamma\,,\ \textit{Service}\mid\textit{Invoker})$$

Soundness

- We resort to declarative type system $\vdash$

- System $\vdash$ relies on non deterministic operation to split contexts

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1, p : S_1 \vdash p : S_1 \qquad \Gamma_2, w : \mathsf{end}, p : S_2 \vdash Q}{\Gamma, p : (S_1, S_2), w : \mathsf{lin}\,!S_1.\mathsf{end} \vdash \overline{w}\langle p \rangle.Q}$$

- typeCheck(Γ, P) implies $\Gamma \vdash P$

Expressivity

- First, we type checked our motivating example

- Still, there are process accepted by $\vdash$ that we do not type check

1. $\Gamma_1, x : (\mathsf{lin}\, ?T.S_1, \mathsf{lin}\, !T.S_2) \vdash \overline{x}\langle v\rangle.C[x(y).P]$

2. $\Gamma_2, x : (\mathsf{lin}\, ?T.S_1, \mathsf{lin}\, !T.S_2) \vdash x(y).C[\overline{x}\langle v\rangle.Q]$

3. $\Gamma_3, x : (\mathsf{lin}\, ?T.S_1, \mathsf{lin}\, !T.S_2) \vdash \overline{x}\langle x\rangle.P$

- But these processes are deadlocked!

- How to prove?

Typed behavioral theory

- In parallel work we proposed typed barbed equivalence for sessions

$$\Delta \models P \cong Q$$

1. $\Gamma_1 \vdash P, \Gamma_2 \vdash Q$
2. Δ *compatible* with Γ_1, Γ_2 (e.g. no interference with a session)
3. P and Q have same barbs in all contexts type checked by Δ

- Proof technique: typed bisimulation
- Technical framework: polyadic pi calculus with matching, meet operation over types...

Application

- Let $\Gamma_i, x : (\mathsf{lin}\, ?T.S_1, \mathsf{lin}\, !T.S_2) \vdash P_i$ for $i = 1, 2, 3$

- Let Δ_i be compatible with $\Gamma_i, x : (\mathsf{lin}\, ?T.S_1, \mathsf{lin}\, !T.S_2)$ for $i = 1, 2, 3$

1. $\Delta_1 \models \overline{x}\langle v\rangle.C[x(y).P] \cong \mathbf{0}$

2. $\Delta_2 \models x(y).C[\overline{x}\langle v\rangle.Q] \cong \mathbf{0}$

3. $\Delta_3 \models \overline{x}\langle x\rangle.P \cong \mathbf{0}$

- Wow! So, what?

Towards semantic completeness

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

- Construction: take derivation tree for $\Gamma_1 \vdash P_1$ and substitute each occurrence of $\Gamma, x : (\text{lin}\ ?T.S_1, \text{lin}\ !T.S_2) \vdash \overline{x}\langle v\rangle.Q$ with $\emptyset \vdash \mathbf{0}$

- Typed equivalence: Δ compatible with Γ_1, Γ_2 implies

$$\Delta \models P_1 \cong P_2$$

- Semantic completeness (in progress):

  If $\Gamma_1 \vdash P_1$ *with* Γ_1 *balanced, then there is a transformation* $\Gamma_2 \vdash P_2$ *such that* $\Delta \models P_1 \cong P_2$ *and* $\text{typeCheck}\,(\Gamma_2, P_2)$ *.*

Conclusions

- We introduced a type checking algorithm for the analysis of structured-oriented programs in the pi calculus

- Technique relies on construct that describes the two ends of the same channel
  - Each end point is a linear or an unrestricted session type
  - Linear types evolve to unrestricted types

- The algorithm is sound w.r.t. type system $\vdash$

- We claim to type check all interesting processes accepted by $\vdash$

Usefulness

- System $\vdash$ enjoys type-preserving encodings of
  1. linear lambda calculus [Walker&05]
  2. linear pi calculus [KPT&TOPLAS'99]
  3. pi calculus with polarities [GH&Acta'05]
- We therefore offer an algorithm for typing functional and mobile languages based on linearity
- Other systems can be considered

$$[\![(\nu xy\colon S)P]\!] = (\nu x\colon (S, \overline{S}))[\![P[x/y]]\!]$$ [V@SFM'09]

Ongoing and future work

- Semantic completeness in progress

- Still, there are interesting processes that are not typable by $\vdash$

$$!x(y).(\nu a)(\overline{y}\langle a\rangle.a(\mathsf{title}).a(\mathsf{date}).(!a(\mathsf{date}) \mid \overline{a}\langle \mathsf{22March}\rangle))$$

- Both capabilities needed in continuation for receive and send date

- Sub typing à la Pierce&Sangiorgi would fix this