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 = \text{un} $?T.S_1$ unrestricted recursive input
  $S_2 = \text{un} !T.S_2$ 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

\[ \overline{\text{poll}}\langle y \rangle . y(p) . (\overline{p}\langle \text{Workshop} \rangle . \overline{p}\langle 9\text{June} \rangle . (\overline{z_1}\langle p \rangle | \cdots | \overline{z_n}\langle p \rangle ))) \]

- 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)

\[
\bar{p}\langle \text{Workshop}\rangle . \bar{p}\langle 9\text{June}\rangle . (\bar{z}_1\langle p\rangle \mid \cdots \mid \bar{z}_n\langle p\rangle)
\]

- End point session type for channel \( p \) is
  \( \text{lin !string} . \text{lin !date} . \ast\text{un !date} \)

- Recursive type \( \ast\text{un !date} \) allows for distribution of poll channel
Concurrent session types

- Service: instantiation generates poll
  \[
  \text{Service} = !\text{poll}(w). (\nu p : T) \overline{w}(p). p(\text{title}). p(\text{date}). !p(\text{date})
  \]

- One channel end sent to the invoker
  \[
  S_1 = \text{lin} !\text{string}. \text{lin} !\text{date}. * \text{un} !\text{date}
  \]

- The other channel end used in the continuation
  \[
  S_2 = \text{lin} ?\text{string}. \text{lin} ?\text{date}. * \text{un} ?\text{date}
  \]

- Full type for \( p \) describes \textit{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 \( P | Q \) 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

\[
(\sim ML) \quad \text{fun typeVar}(\Gamma, x : (\text{lin}\, T_1 \cdot S_1, \text{lin}\, !T_2 \cdot S_2), x : \text{lin}\, ?T_1 \cdot S_1) = \Gamma, x : (\circ, \text{lin}\, !T_2 \cdot S_2)
\]
ML patterns

• Typings for processes are patterns of function:
  \[
  \text{fun check}(g : \text{context}, p : \text{process}) : \text{context}
  \]

• Patterns matching deterministic for safe types (no backtracking)
  1. dual unrestricted channel ends: (\(*\text{un}?T, *\text{un}!T\)) and \(T\) safe
  2. dual linear ends: (lin?\(T.S_1\), lin!\(T.S_2\)) and \(T\) and \((S_1, S_2)\) safe

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

  \[
  \text{check}(\Gamma, x : (\text{lin}?T.S_1, \text{lin}!T.S_2), x(y).P) = \\
  \text{let val d } = \text{check}(\Gamma, x : S_1, y : T, P) \text{ in } ...
  \]
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

\[
P = x(y).if\ y\ then\ 0\ else\ 0\ |\ (\nu z : end)x(z).0
\]

\[
x : (\text{lin? bool.end, lin! end.end}) \vdash P
\]

\[
P \rightarrow (\nu z : end)\text{if}\ z\ then\ 0\ else\ 0
\]

\[
\not\vdash (\nu z : end)\text{if}\ z\ then\ 0\ else\ 0
\]

- 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 $\circ, \text{un } p, (\text{un } p_1, \text{un } p_2), (\text{un } p, \circ), (\circ, \circ)$

fun typeCheck($\Gamma : \text{context}, P : \text{process} ) : \text{bool} =$

if safe($\Gamma$) then

let val $\Delta =$ check($\Gamma, P$) in

if consumed($\Delta$) then true
A run

- Protocol described by concurrent execution of

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

\[
\text{Invoker} = \overline{\text{poll}} \langle y \rangle . y(p). (\overline{p} \langle \text{Workshop} \rangle. \overline{p} \langle 9\text{June} \rangle. (\overline{z_1} \langle p \rangle | .. | \overline{z_n} \langle p \rangle))
\]

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

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

\[
T = (S_1, S_2)
\]

\[
T_w = \text{lin}!S_1. \text{un end}
\]

- Type checking succeeds

\[
\text{typeCheck}(\Gamma, \text{poll : } (*\text{un} ?T_w, *\text{un} !T_w) , \text{Service} | \text{Invoker})
\]
Checking the service continuation

- Replicated input spawns a thread for the poll

\[
Service = !C
\]
\[
C = \text{poll}(w). (\nu p : T) \overline{w(p)}. p(\text{title}). p(\text{date}). !p(\text{date})
\]

- Call requires environment in output = environment in input

- Intuition: only linear types change!

\[
\text{check}(\Gamma, Service) = \\
\text{let val } \Delta = \text{check}(\Gamma, C)\text{in} \\
\text{if } (\Delta = \Gamma) \text{ then } \Delta
\]
Checking unrestricted input

- Service instantiation generates poll

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

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

\[
T_w = \text{lin!} S_1.\text{un end}
\]

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

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

\[
\text{check}(\Gamma, \text{poll} : (\ast \text{un ?} T_w, \ast \text{un !} T_w), C) =
\]

\[
\text{let val } \Delta = \text{check}(\Gamma, \text{poll} : (\ast \text{un ?} T_w, \ast \text{un !} T_w), w : T_w, C') \text{ in}
\]

\[
\text{if } (\Delta = \Delta', w : S) \text{ and } (S = \circ \text{ or } S = \text{un } p) \text{ then } \Delta'
\]
Checking restriction

• A poll with safe channel type is generated

\[ C' = (\nu p : (S_1, S_2))w(q).Q \]

\[ \Omega = \Gamma, \text{poll} : (*\text{un} ?T_w, *\text{un} !T_w), w : T_w \]

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

\[
\text{check}(\Omega, C') = \\
\text{if } \text{safe}((S_1, S_2)) \text{ then} \\
\text{let } \text{val } \Delta = \text{check}(\Omega, p : (S_1, S_2), \bar{w}(q).Q) \text{ in} \\
\text{if } (\Delta = \Delta', p : (S', S'')) \text{ then } \Delta' \\
\text{and } (S' = \circ \text{ or } S' = \text{un} p) \\
\text{and } (S'' = \circ \text{ or } S'' = \text{un} p) \\
\text{then } \Delta'
\]
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

\[ P = \overline{\langle p \rangle}_{1} | \overline{\langle p \rangle}_{2} | \cdots | \overline{\langle p \rangle}_{n} \]

\[ S = \ast \text{un} ! \text{date} \]

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

- Parallel processes typed compositionally

\[
\text{check}(\Gamma, P) = \\
\text{let val } \Delta = \text{check}(\Gamma, \overline{\langle p \rangle}_{1}) \text{ in} \\
\text{check}(\Delta, \overline{\langle p \rangle}_{2} | \cdots | \overline{\langle p \rangle}_{n})
\]

- In each call, the type of \( p \) in the input environment is \((\circ, S)\)
Exchanging compositions’ order

- Type checking the service protocol

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

- Preservation of structural congruence!

\[
\text{check}(\Gamma, \text{Invoker} \mid \text{Service}) = \text{check}(\Gamma, \text{Service} \mid \text{Invoker})
\]
Soundness

- We resort to declarative type system $\vdash$

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

$\Gamma = \Gamma_1 \cdot \Gamma_2$
$\Gamma_1, p : S_1 \vdash p : S_1$
$\Gamma_2, w : \text{end}, p : S_2 \vdash Q$

$\Gamma, p : (S_1, S_2), w : \text{lin}\! S_1.\text{end} \vdash \overline{w}\langle p \rangle.Q$

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

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

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

- But these processes are deadlockened!

- How to prove?
Typed behavioral theory

- In parallel work we proposed typed barbed equivalence for sessions

\[ \Delta \models P \equiv Q \]

1. \( \Gamma_1 \vdash P, \Gamma_2 \vdash Q \)

2. \( \Delta \) compatible with \( \Gamma_1, \Gamma_2 \) (e.g. no interference with a session)

3. \( P \) and \( Q \) have same barbs in all contexts type checked by \( \Delta \)

- Proof technique: typed bisimulation

- Technical framework: polyadic pi calculus with matching, meet operation over types...
• Let $\Gamma_i, x : (\text{lin } ?T.S_1, \text{lin } !T.S_2) \vdash P_i$ for $i = 1, 2, 3$

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

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

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

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

• 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 $\Gamma, x : (\text{lin } ?T.S_1, \text{lin } !T.S_2) \vdash x\langle v \rangle.Q$ with $\emptyset \vdash 0$

- Typed equivalence: $\Delta$ compatible with $\Gamma_1, \Gamma_2$ implies
  \[ \Delta \models P_1 \simeq P_2 \]

- Semantic completeness (in progress):
  If $\Gamma_1 \vdash P_1$ with $\Gamma_1$ balanced, then there is a transformation $\Gamma_2 \vdash P_2$ such that $\Delta \models P_1 \simeq P_2$ and 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 $\not\vdash$
Usefulness

- System ⊢ 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

\[
\left[(\nu xy: S)P\right] = (\nu x: (S, \overline{S}))\left[P[x/y]\right]
\]

[V@SFM’09]
Ongoing and future work

- Semantic completeness in progress

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

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

- Both capabilities needed in continuation for receive and send date

- Sub typing à la Pierce&Sangiorgi would fix this