Types and deadlock freedom in a calculus of services, sessions and pipelines

Roberto Bruni[1] Leonardo Gaetano Mezzina[2]

[1]Dipartimento di Informatica
Università di Pisa

[2]IMT Lucca
Instute for Advanced Studies

AMAST 2008
Urbana-Champaign, IL, USA
28–31 July 2008

Outline

Outline

Service Oriented Computing (SOC)

Services

SOC is an emerging paradigm where services are understood as

- autonomous
- platform-independent

computational entities that can be:

- described
- published
- discovered
- dynamically assembled

for developing massively distributed, interoperable, evolvable systems.

e-Expectations

Big companies put many efforts for service delivery on a variety of computing platforms.
Tomorrow, there will be a plethora of new services for e-business, e-health, e-government, e-* within the rapidly evolving Information Society.

Semantic foundations?

Industrial consortia are developing orchestration and choreography languages, targeting the standardization of Web Services and XML-centric technologies for which *neat semantic foundations are necessary*.

Service Oriented Computing (SOC)

Services

SOC is an emerging paradigm where services are understood as

- autonomous
- platform-independent

computational entities that can be:

- described
- published
- discovered
- dynamically assembled

for developing massively distributed, interoperable, evolvable systems.

e-Expectations

Big companies put many efforts for service delivery on a variety of computing platforms.
Tomorrow, there will be a plethora of new services for e-business, e-health, e-government, e-* within the rapidly evolving Information Society.

Semantic foundations?

Industrial consortia are developing orchestration and choreography languages, targeting the standardization of Web Services and XML-centric technologies for which *neat semantic foundations are necessary*.

SENSORIA (http://www.sensoria-ist.eu)

IST-FET Integrated Project funded by the EU in the GC Initiative (6th FP).

SENSORIA

Software Engineering for Service-Oriented Overlay Computers

Aim

Developing a novel, comprehensive approach to the engineering of software systems for service-oriented overlay computers.

Strategy

Integration of foundational theories, techniques, methods and tools in a pragmatic software engineering approach.

The role of process calculi

Coordinating and combining services

A crucial role in the project is played by formalisms for service description that can lay the mathematical basis for analysing and experimenting with components interactions, and for combining services.

SENSORIA Work Package 2

We seek for a small set of primitives that might serve as a basis for formalizing, programming and disciplining service oriented applications over global computers.

SENSORIA core calculi

- *Signal Calculus*: middleware level
- *SOCK, COWS*: service level, correlation-based
- *SCC-family (**SCC**, SSCC, CC, CaSPiS)*: service level, session-based
- *cc-pi, lambda-req*: SLA contract level

Main Contribution

Goal

Define a type system for SCC to guarantee sound interaction.

Proceedings

- Syntax + LTS semantics (see Section 2)
- Type system + subject reduction (see Section 3)
- **Initial processes are deadlock free**: We define a class of processes, called *initial*, for which we can guarantee that a normal form is reached with no pending session protocols unless infinitely many services are invoked provoking divergence (see Theorem 2).
- Simple examples

Talk

- Sketches of syntax and semantics
- Intuitive idea and flashes of typing rules
- Simple examples

Related Work

- Honda, Vasconcelos, Kubo + Gay, Hole + Kobayashi: starting point
- Acciai, Boreale (Ugo Montanari's Festschrift): CaSPiS^{-}, asymmetric notion of progress
- Dezani et al. (TGC'07): progress, no recursion
- Lanese et al. (SEFM'07): SSCC orchestration is via streams instead of pipelines
- Bruni et al. (PLACES'08): μse, dynamic multiparty sessions
- Bonelli, Compagnoni + Honda, Yoshida, Carbone: multiparty asynchronous sessions
- ...

Outline

SCC Genesis

SCC [WS-FM 2006] sources of inspiration

- π (names, communication): $x(y).P$, $\overline{x}y.P$, $(\nu x)P$
- πI, structured communication (session types): $a(k).P$, $\overline{a}(k).P$
  roughly, think of $\overline{a}(k).P$ as $(\nu k)\overline{a}k.P$
- Orc (pipelining and pruning of activities):
  $(EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle) > cfp > Email\langle \texttt{rb@gmail.it}, cfp\rangle$
  $Email\langle \texttt{rb@gmail.it}, cfp\rangle$ **where** $cfp :\in (EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle)$

To keep in mind

We are dealing with conceptual abstractions: the syntax does not necessarily expose implementation details. For example:

- a session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging
- all service instances (serving different requests) can be handled by one service port

SCC Genesis

SCC [WS-FM 2006] sources of inspiration

- π (names, communication): $x(y).P$, $\overline{x}y.P$, $(\nu x)P$
- πI, structured communication (session types): $a(k).P$, $\overline{a}(k).P$
  roughly, think of $\overline{a}(k).P$ as $(\nu k)\overline{a}k.P$
- Orc (pipelining and pruning of activities):
  $(EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle) > cfp > Email\langle \texttt{rb@gmail.it}, cfp\rangle$
  $Email\langle \texttt{rb@gmail.it}, cfp\rangle$ **where** $cfp :\in (EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle)$

To keep in mind

We are dealing with conceptual abstractions: the syntax does not necessarily expose implementation details. For example:

- a session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging
- all service instances (serving different requests) can be handled by one service port

SCC Genesis

SCC [WS-FM 2006] sources of inspiration

- π (names, communication): $x(y).P$, $\overline{x}y.P$, $(\nu x)P$
- πI, structured communication (session types): $a(k).P$, $\overline{a}(k).P$
  roughly, think of $\overline{a}(k).P$ as $(\nu k)\overline{a}k.P$
- Orc (pipelining and pruning of activities):
  $(EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle) > cfp > Email\langle \texttt{rb@gmail.it}, cfp\rangle$
  $Email\langle \texttt{rb@gmail.it}, cfp\rangle$ **where** $cfp :\in (EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle)$

To keep in mind

We are dealing with conceptual abstractions: the syntax does not necessarily expose implementation details. For example:

- a session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging
- all service instances (serving different requests) can be handled by one service port

SCC Genesis

SCC [WS-FM 2006] sources of inspiration

- π (names, communication): $x(y).P$, $\overline{x}y.P$, $(\nu x)P$
- πI, structured communication (session types): $a(k).P$, $\overline{a}(k).P$
  roughly, think of $\overline{a}(k).P$ as $(\nu k)\overline{a}k.P$
- Orc (pipelining and pruning of activities):
  $(EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle) > cfp > Email\langle \texttt{rb@gmail.it}, cfp\rangle$
  $Email\langle \texttt{rb@gmail.it}, cfp\rangle$ **where** $cfp :\in (EAPLS\langle 2008\rangle \mid EATCS\langle 2008\rangle)$

To keep in mind

We are dealing with conceptual abstractions: the syntax does not necessarily expose implementation details. For example:

- a session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging
- all service instances (serving different requests) can be handled by one service port

SCC: General Principles

Service definitions: $s.P$

- services expose their protocols
- (persistent) services can handle multiple requests separately

Service invocations: $\overline{s}.P$

- service invocations expose their protocols
- sequential composition via pipelining (á la Orc)

Sessions: $r \triangleright P$

- seen as run-time syntax
- service invocation spawns fresh session parties (locally to each partner)
- sessions are: two-party (service-side + client-side) + private
- interaction between session protocols: bi-directional
- nested sessions: values can be returned outside sessions (one level up)

SCC: General Principles

Service definitions: $s.P$

- services expose their protocols
- (persistent) services can handle multiple requests separately

Service invocations: $\overline{s}.P$

- service invocations expose their protocols
- sequential composition via pipelining (á la Orc)

Sessions: $r \triangleright P$

- seen as run-time syntax
- service invocation spawns fresh session parties (locally to each partner)
- sessions are: two-party (service-side + client-side) + private
- interaction between session protocols: bi-directional
- nested sessions: values can be returned outside sessions (one level up)

SCC: General Principles

Service definitions: $s.P$

- services expose their protocols
- (persistent) services can handle multiple requests separately

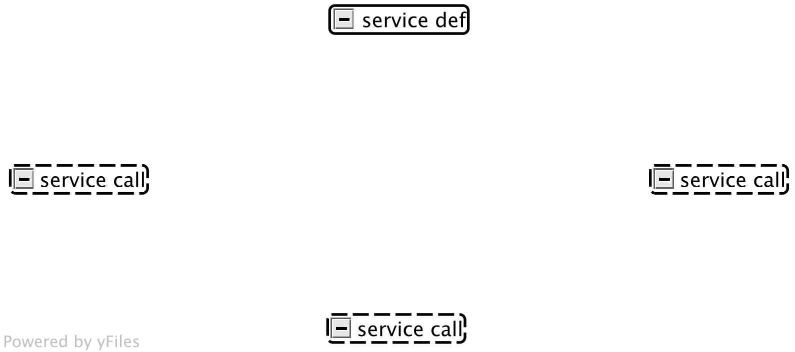
Service invocations: $\overline{s}.P$

- service invocations expose their protocols
- sequential composition via pipelining (á la Orc)

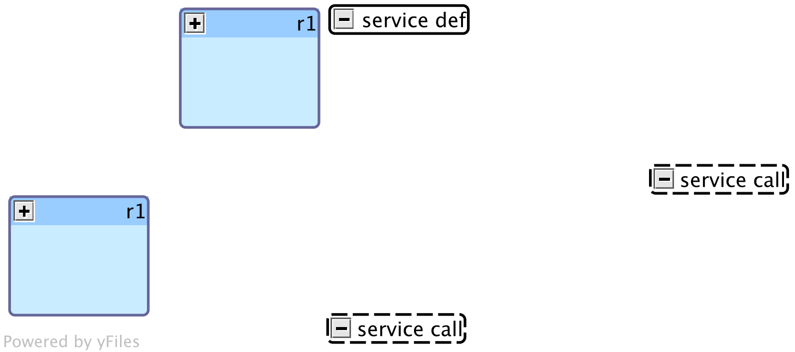
Sessions: $r \triangleright P$

- seen as run-time syntax
- service invocation spawns fresh session parties (locally to each partner)
- sessions are: two-party (service-side + client-side) + private
- interaction between session protocols: bi-directional
- nested sessions: values can be returned outside sessions (one level up)

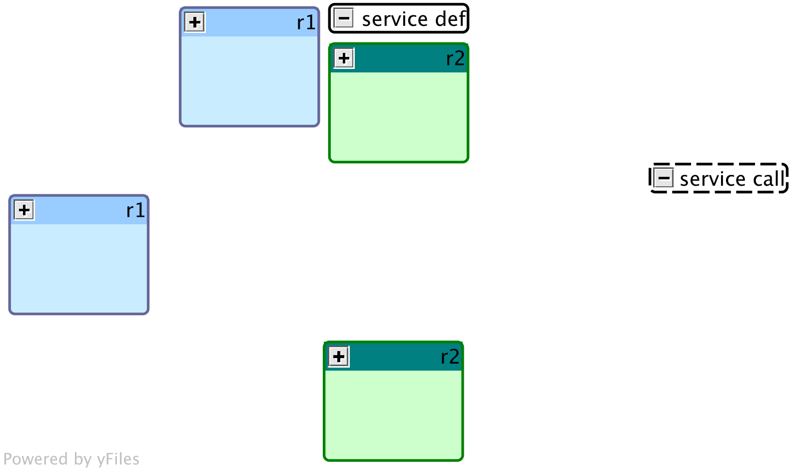
Sketch of Multiple Sessions



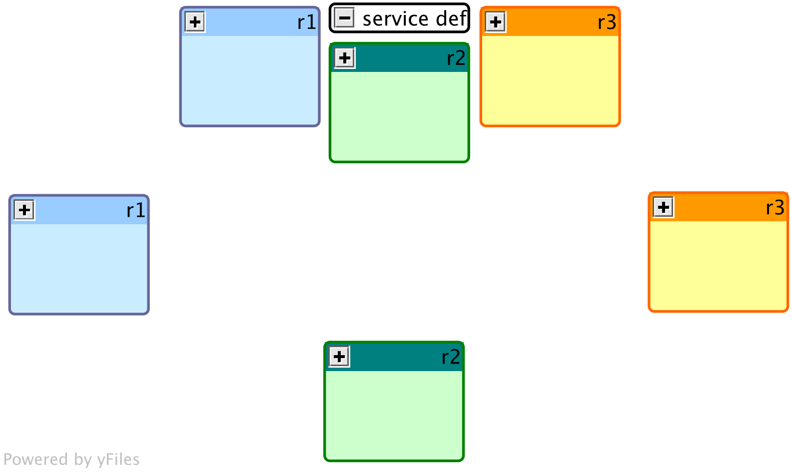
Sketch of Multiple Sessions



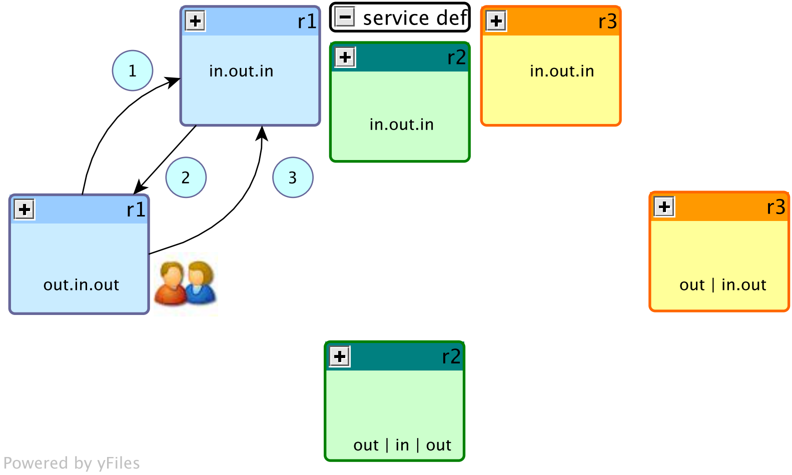
Sketch of Multiple Sessions



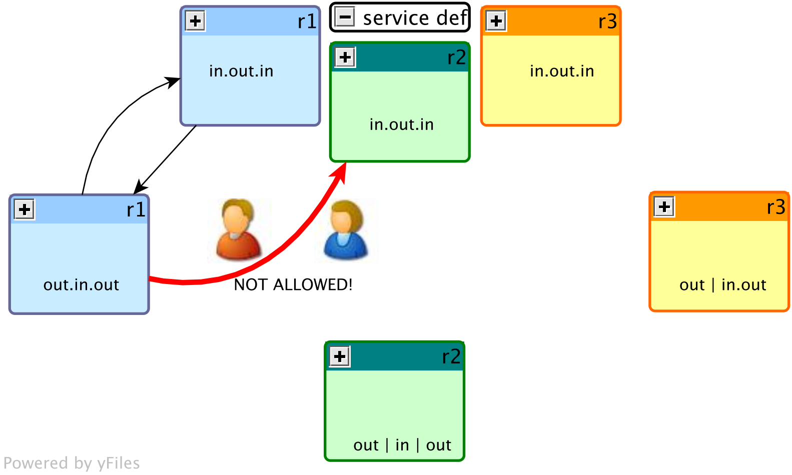
Sketch of Multiple Sessions



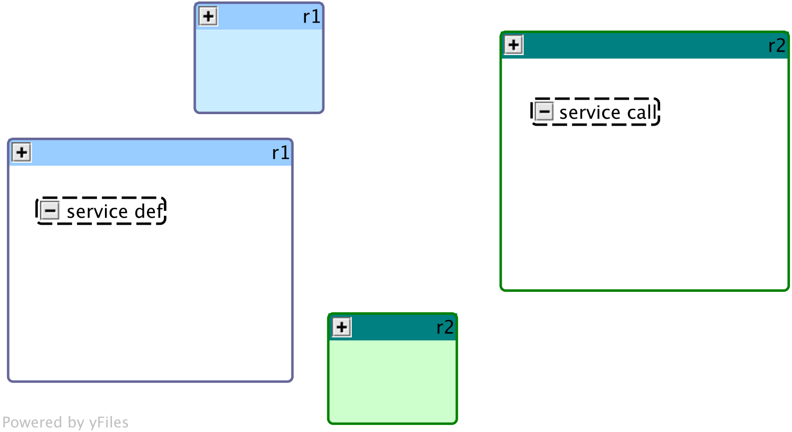
Sketch of Conversations



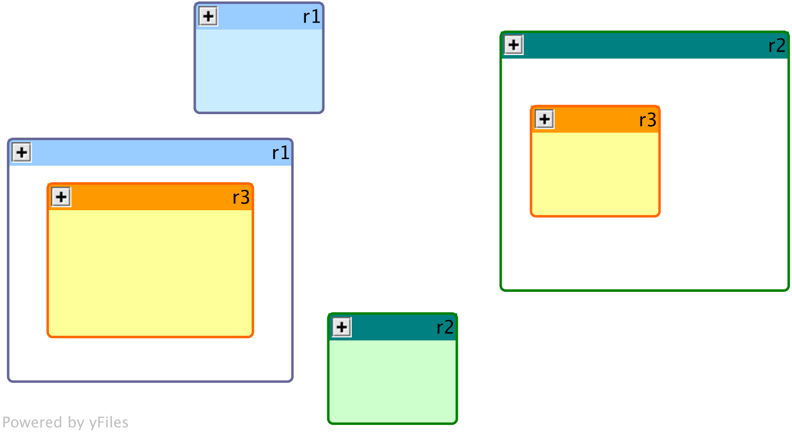
Sketch of Conversations



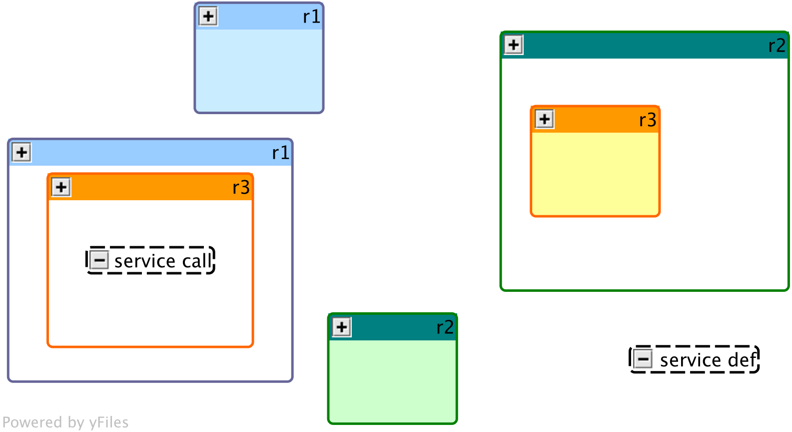
Sketch of Nested Sessions



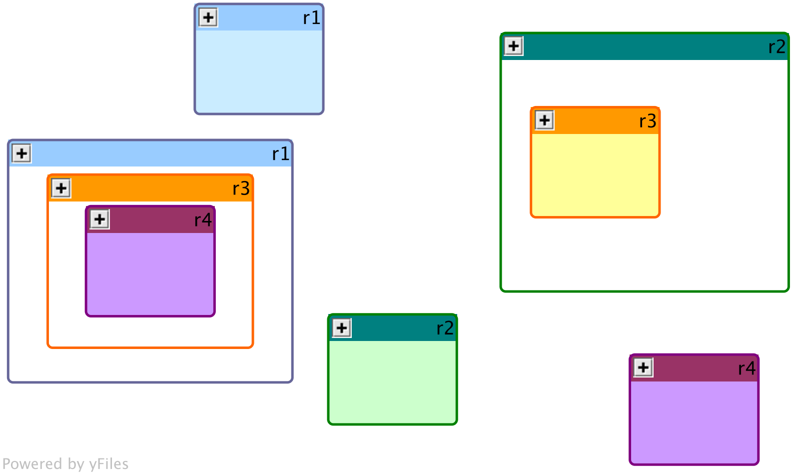
Sketch of Nested Sessions



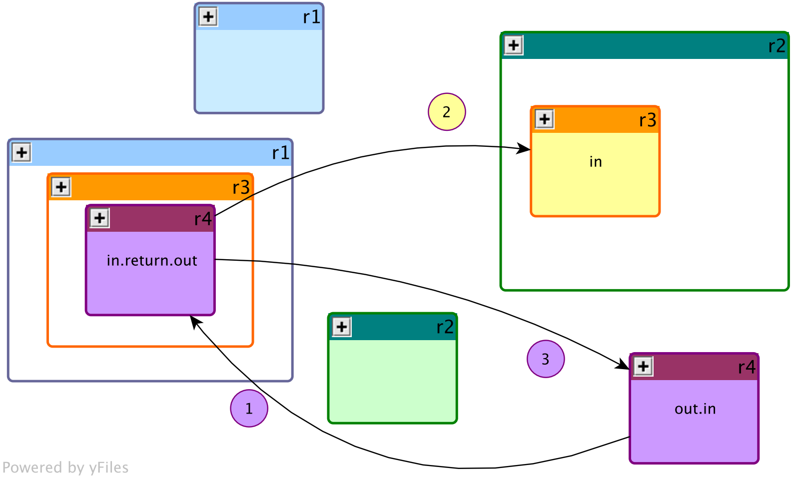
Sketch of Nested Sessions



Sketch of Nested Sessions



Sketch of Return



SCC Raw Syntax

Names, Values, Polarities

$$
\begin{array}{rcll}
m & ::= & s \mid r & \text{(name)} \\
v & ::= & b \mid s \mid x \mid f(\tilde{v}) & \text{(value)} \\
p, q & ::= & - \mid + & \text{(polarity)}
\end{array}
$$

Processes

$$
\begin{array}{rllll}
P, Q ::= & & & \mathbf{0} & \text{(nil)} \\
& \mid & s.P & \mid \ \overline{v}.Q & \text{(service definition / invoke)} \\
& \mid & \langle \tilde{v} \rangle.P & \mid \ (\tilde{x}).Q & \text{(values output / tuple input)} \\
& \mid & \langle l \rangle.P & \mid \ \Sigma_{i=1}^{n}(l_i).P_i & \text{(label selection / branching)} \\
& & & \mid \ \texttt{return}\ \tilde{v}.P & \text{(return)} \\
& & & \mid \ \texttt{if}\ v = v'\ \texttt{then}\ P\ \texttt{else}\ Q & \text{(if-then-else)} \\
& & & \mid \ (\nu m)P & \text{(restriction)} \\
& & & \mid \ r^{p} \rhd P & \text{(session)} \\
& & & \mid \ P > \tilde{x} > Q & \text{(pipe)} \\
& & & \mid \ P|Q & \text{(parallel)}
\end{array}
$$

SCC Raw Syntax

Names, Values, Polarities

$$
\begin{array}{rcll}
m & ::= & s \mid r & \text{(name)} \\
v & ::= & b \mid s \mid x \mid f(\tilde{v}) & \text{(value)} \\
p, q & ::= & - \mid + & \text{(polarity)}
\end{array}
$$

Processes

$$
\begin{array}{llllll}
P, Q & ::= & & & \mathbf{0} & \text{(nil)} \\
& \mid & s.P & \mid & \overline{v}.Q & \text{(service definition / invoke)} \\
& \mid & \langle \tilde{v} \rangle.P & \mid & (\tilde{x}).Q & \text{(values output / tuple input)} \\
& \mid & \langle l \rangle.P & \mid & \Sigma_{i=1}^{n}(l_i).P_i & \text{(label selection / branching)} \\
& & & \mid & \texttt{return } \tilde{v}.P & \text{(return)} \\
& & & \mid & \texttt{if } v = v' \texttt{ then } P \texttt{ else } Q & \text{(if-then-else)} \\
& & & \mid & (\nu m)P & \text{(restriction)} \\
& & & \mid & r^{p} \triangleright P & \text{(session)} \\
& & & \mid & P > \tilde{x} > Q & \text{(pipe)} \\
& & & \mid & P|Q & \text{(parallel)}
\end{array}
$$

SCC Structural Congruence

Axioms

- alpha-conversion
- parallel composition
- name restriction
- garbage collection of terminated sessions

SCC Structural Congruence

Standard axioms (assume $m, y \notin \mathrm{fn}(Q)$ and $r \neq m$)

$$(\nu m')Q \equiv (\nu m)(Q[^m/_{m'}]) \qquad (\tilde{x}).Q \equiv\equiv (\tilde{y}).Q[^y/_x]$$
$$P > \tilde{x} > Q \equiv P > \tilde{y} > (Q[^y/_x])$$

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R)$$

$$Q|((\nu m)P) \equiv (\nu m)(Q|P) \qquad (\nu m)(\nu m')P \equiv (\nu m')(\nu m)P$$
$$r^p \triangleright (\nu m)P \equiv (\nu m)(r^p \triangleright P) \qquad ((\nu m)P) > \tilde{x} > Q \equiv (\nu m)(P > \tilde{x} > Q)$$

Axioms for garbage collection of terminated sessions

$$\mathbf{0} > \tilde{x} > P \equiv \mathbf{0} \qquad (P|Q) > \tilde{x} > R \equiv (P > \tilde{x} > R)|(Q > \tilde{x} > R)$$

$$(r^p \triangleright \mathbf{0}) > \tilde{x} > R \equiv r^p \triangleright \mathbf{0} \qquad r_1^p \triangleright (Q|r_2^q \triangleright \mathbf{0}) \equiv r_1^p \triangleright Q|r_2^q \triangleright \mathbf{0}$$

$$(\nu r)(r^+ \triangleright \mathbf{0}|r^- \triangleright \mathbf{0}) \equiv \mathbf{0}$$

SCC Fragment

Main assumptions

Services are

- **persistent** (not consumed after invocations)
- **top-level** (not nested, not dynamically installed)
- **stateless** (returns not allowed on service side)

Sessions are

- not interruptable (**close-free** fragment)
- with **non recursive** communication protocols

Example 1: Factorial

Service definition

$$\begin{aligned} fatt.(n).\texttt{if } & (n = 0) \\ & \texttt{then } \langle 1 \rangle \\ & \texttt{else } (\overline{fatt}.\langle n-1 \rangle.(x).\texttt{return } x) > x > \langle n \cdot x \rangle \end{aligned}$$

A *fatt* instance waits for a natural number n: if equal to zero then sends back 1 to the client, otherwise issues an invocation to a fresh instance of *fatt* with argument $n-1$, waits for the response and passes the result x to a pipe that sends back $n \cdot x$ to the client

Service invocation

$$\overline{fatt}.\langle 3 \rangle.(x) \quad | \quad \overline{fatt}.\langle 5 \rangle.(x).\texttt{return } x$$

The first client passes the argument 3 to the service instance, then waits for the response; the second client passes a different argument and returns the computed result to the parent session. The protocols of the two clients will run in separate sessions and will not interfere.

Example 1: Factorial

Service definition

$$\begin{aligned} \mathit{fatt}.(n).\texttt{if } & (n = 0) \\ & \texttt{then } \langle 1 \rangle \\ & \texttt{else } (\overline{\mathit{fatt}}.\langle n-1 \rangle.(x).\texttt{return } x) > x > \langle n \cdot x \rangle \end{aligned}$$

A *fatt* instance waits for a natural number n: if equal to zero then sends back 1 to the client, otherwise issues an invocation to a fresh instance of *fatt* with argument $n-1$, waits for the response and passes the result x to a pipe that sends back $n \cdot x$ to the client

Service invocation

$$\overline{\mathit{fatt}}.\langle 3 \rangle.(x) \quad | \quad \overline{\mathit{fatt}}.\langle 5 \rangle.(x).\texttt{return } x$$

The first client passes the argument 3 to the service instance, then waits for the response; the second client passes a different argument and returns the computed result to the parent session. The protocols of the two clients will run in separate sessions and will not interfere.

Example 2: Room reservation

Service definition (with branching)

$$\mathit{reserve}.\Big(\ (\text{single}).(x).\langle \text{code}(x,\text{""})\rangle \\ + (\text{double}).(x,y).\langle \text{code}(x,y)\rangle\ \Big)$$

(where code : $str \times str \rightarrow int$ is a function only available on service side)

Service invocations (with selection)

$\overline{\mathit{reserve}}.\langle \text{single}\rangle.\langle \texttt{"Bob"}\rangle.(x).\texttt{return}\ x$

$\overline{\mathit{reserve}}.\langle \text{double}\rangle.\langle \texttt{"Bob"},\texttt{"Leo"}\rangle.(y).\texttt{return}\ y$

$\overline{\mathit{reserve}}.\texttt{if}\ (\ldots)$
$\quad \texttt{then}\ \langle \text{single}\rangle.\langle \texttt{"Bob"}\rangle.(x).\texttt{return}\ x$
$\quad \texttt{else}\ \langle \text{double}\rangle.\langle \texttt{"Bob"},\texttt{"Leo"}\rangle.(y).\texttt{return}\ y$

Example 2: Room reservation

Service definition (with branching)

$$reserve.\Big(\ (\text{single}).(x).\langle\text{code}(x, \text{""})\rangle \\ + (\text{double}).(x, y).\langle\text{code}(x, y)\rangle \Big)$$

(where code : $str \times str \to int$ is a function only available on service side)

Service invocations (with selection)

$$\overline{reserve}.\langle\text{single}\rangle.\langle\text{"Bob"}\rangle.(x).\texttt{return}\ x$$

$$\overline{reserve}.\langle\text{double}\rangle.\langle\text{"Bob"},\text{"Leo"}\rangle.(y).\texttt{return}\ y$$

$$\overline{reserve}.\texttt{if}\ (\ldots) \\ \texttt{then}\ \langle\text{single}\rangle.\langle\text{"Bob"}\rangle.(x).\texttt{return}\ x \\ \texttt{else}\ \langle\text{double}\rangle.\langle\text{"Bob"},\text{"Leo"}\rangle.(y).\texttt{return}\ y$$

Example 3: Proxy service for load balancing

Service definition (with name passing and extrusion)

$$(\nu a, b)\Big(\ a.P \\ \mid b.P \\ \mid \mathit{loadbalance}.\mathtt{if}\ (\mathrm{choose}(a, b) = 1)\ \mathtt{then}\ \langle a \rangle\ \mathtt{else}\ \langle b \rangle\ \Big)$$

Service invocation

$$(\overline{\mathit{loadbalance}}.(z).\mathtt{return}\ z) > x > \overline{z}.Q$$

Example 3: Proxy service for load balancing

Service definition (with name passing and extrusion)

$$
\begin{aligned}
(\nu a, b)\Big(\ & a.P \\
\mid\ & b.P \\
\mid\ & \mathit{loadbalance}.\mathtt{if}\ (\mathrm{choose}(a, b) = 1)\ \mathtt{then}\ \langle a \rangle\ \mathtt{else}\ \langle b \rangle\ \Big)
\end{aligned}
$$

Service invocation

$$(\overline{\mathit{loadbalance}}.(z).\mathtt{return}\ z) > x > \overline{z}.Q$$

Outline

Type judgements

Overall idea

- Type values: $\Gamma \vdash v : S$
- Type a process as if part of a current session:

$$\Gamma \vdash P : U[T]$$

  separating intra-session interaction T from upward interaction U
- The type T of the protocol on one side of a session should be dual w.r.t. the type T' of its partner's protocol ($\overline{T} = T'$)

Problems (and limitations)

- Some flexibility required w.r.t. branching and selection
- Parallel composition of protocols
- Replication due to pipelines
- No delegation
- Recursive invocation of services

Type system basics

Syntax of types

$$
\begin{array}{rcll}
S & ::= & [T] & \text{(session)} \\
 & | & \mathcal{B} & \text{(basic data types)} \\
T & ::= & \mathtt{end} & \text{(no action)} \\
 & | & ?(S_1, \ldots, S_n).T & \text{(input of a tuple)} \\
 & | & !(S_1, \ldots, S_n).T & \text{(output of a tuple)} \\
 & | & \&\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(external choice)} \\
 & | & \oplus\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(internal choice)} \\
U & ::= & !(\tilde{S})^k.\mathtt{end} & \text{(upward interaction)}
\end{array}
$$

Dual types

$$
\overline{\mathtt{end}} = \mathtt{end} \qquad
\begin{array}{rcl}
\overline{?(\tilde{S}).T} & = & !(\tilde{S}).\overline{T} \\
\overline{!(\tilde{S}).T'} & = & ?(\tilde{S}).\overline{T'}
\end{array}
\qquad
\begin{array}{rcl}
\overline{\&\{l_i : T_i\}_i} & = & \oplus\{l_i : \overline{T_i}\}_i \\
\overline{\oplus\{l_i : T_i\}_i} & = & \&\{l_i : \overline{T_i}\}_i
\end{array}
$$

Type System Highlights: Services and Sessions

Services

(Service)
$$\Gamma, s : S \vdash s : S$$

(Tdef)
$$\frac{\Gamma \vdash P : \mathtt{end}[T] \quad \Gamma \vdash s : [T]}{\Gamma \vdash s.P : \mathtt{end}[\mathtt{end}]}$$

(Tinv)
$$\frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash s : [\overline{T}]}{\Gamma \vdash \overline{s}.P : \mathtt{end}[U]}$$

Sessions

(Tses)
$$\frac{\Gamma \vdash P : U[T]}{\Gamma, r : [T] \vdash r^{+} \triangleright P : \mathtt{end}[U]}$$

(TsesI)
$$\frac{\Gamma \vdash P : U[T]}{\Gamma, r : [\overline{T}] \vdash r^{-} \triangleright P : \mathtt{end}[U]}$$

Type System Highlights: Services and Sessions

Services

(Service)

$$\Gamma, s : S \vdash s : S$$

(Tdef)

$$\frac{\Gamma \vdash P : \mathtt{end}[T] \quad \Gamma \vdash s : [T]}{\Gamma \vdash s.P : \mathtt{end}[\mathtt{end}]}$$

(Tinv)

$$\frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash s : [\overline{T}]}{\Gamma \vdash \overline{s}.P : \mathtt{end}[U]}$$

Sessions

(Tses)

$$\frac{\Gamma \vdash P : U[T]}{\Gamma, r : [T] \vdash r^{+} \triangleright P : \mathtt{end}[U]}$$

(TsesI)

$$\frac{\Gamma \vdash P : U[T]}{\Gamma, r : [\overline{T}] \vdash r^{-} \triangleright P : \mathtt{end}[U]}$$

Type System Highlights: Protocols

Input, output, and return

$$\text{(TIn)}\ \frac{\Gamma, \tilde{x} : \tilde{S} \vdash P : U[T]}{\Gamma \vdash (\tilde{x}).P : U[?(\tilde{S}).T]} \qquad \text{(TOut)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma \vdash \langle \tilde{v} \rangle.P : U[!(\tilde{S}).T]}$$

$$\text{(TRet)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma \vdash \mathtt{return}\ \tilde{v}.P : !(\tilde{S}).U[T]}$$

Branching and Selection

$$\text{(TBranch)}\ \frac{I \subseteq \{1, \ldots, n\}\ \forall i \in I. \Gamma \vdash P_i : U[T_i]}{\Gamma \vdash \Sigma_{i=0}^{n}(l_i).P_i : U[\&\{l_i : T_i\}]_{i \in I}} \qquad \text{(TChoice)}\ \frac{k \in I \quad \Gamma \vdash P : U[T_k]}{\Gamma \vdash \langle l_k \rangle.P : U[\oplus\{l_i : T_i\}_{i \in I}]}$$

Type System Highlights: Protocols

Input, output, and return

(TIn)
$$\frac{\Gamma, \tilde{x} : \tilde{S} \vdash P : U[T]}{\Gamma \vdash (\tilde{x}).P : U[?(\tilde{S}).T]}$$

(TOut)
$$\frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma \vdash \langle \tilde{v} \rangle.P : U[!(\tilde{S}).T]}$$

(TRet)
$$\frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma \vdash \texttt{return}\ \tilde{v}.P :!(\tilde{S}).U[T]}$$

Branching and Selection

(TBranch)
$$\frac{I \subseteq \{1, \ldots, n\}\ \forall i \in I. \Gamma \vdash P_i : U[T_i]}{\Gamma \vdash \Sigma_{i=0}^{n}(l_i).P_i : U[\&\{l_i : T_i\}]_{i \in I}}$$

(TChoice)
$$\frac{k \in I \quad \Gamma \vdash P : U[T_k]}{\Gamma \vdash \langle l_k \rangle.P : U[\oplus\{l_i : T_i\}_{i \in I}]}$$

Type System Highlights: Protocols

Parallel

(TparL)

$$\frac{\Gamma \vdash P :!(\tilde{S})^{n}.\mathrm{end}[T] \quad \Gamma \vdash Q :!(\tilde{S})^{m}.\mathrm{end}[\mathrm{end}]}{\Gamma \vdash P|Q :!(\tilde{S})^{n+m}.\mathrm{end}[T]}$$

Conditional

(Tif)

$$\frac{\Gamma \vdash v_1 : S \quad \Gamma \vdash v_2 : S \quad \Gamma \vdash P : U[T] \quad \Gamma \vdash Q : U[T]}{\Gamma \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : U[T]}$$

Type System Highlights: Protocols

Parallel

(TParL)

$$\frac{\Gamma \vdash P :!(\tilde{S})^{n}.\mathtt{end}[T] \quad \Gamma \vdash Q :!(\tilde{S})^{m}.\mathtt{end}[\mathtt{end}]}{\Gamma \vdash P|Q :!(\tilde{S})^{n+m}.\mathtt{end}[T]}$$

Conditional

(Tif)

$$\frac{\Gamma \vdash v_1 : S \quad \Gamma \vdash v_2 : S \quad \Gamma \vdash P : U[T] \quad \Gamma \vdash Q : U[T]}{\Gamma \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : U[T]}$$

Main properties

Subject Congruence

If $\Gamma \vdash P : U[T]$ and $P \equiv Q$ then $\Gamma \vdash Q : U[T]$

Subject reduction

- If $\Gamma, r : S \vdash P : U[T]$ and $P \xrightarrow{r\tau} Q$ then $\Gamma, r : S' \vdash Q : U[T]$
- If $\Gamma \vdash P : U[T]$ and $P \xrightarrow{\tau} Q$ then $\Gamma \vdash Q : U[T]$

- $P \xrightarrow{r\tau} Q$ means that Q is reached by P after a communication or a selection within session r, with r a free name in P
- $P \xrightarrow{\tau} Q$ means that Q is reached by P after interaction in a restricted session or after a service invocation

Main result

Normal form

$P \equiv (\nu s_1) \ldots (\nu s_n)(s_1.Q_1| \ldots |s_n.Q_n)$

Deadlock free processes

P such that whenever $P \xrightarrow{\omega}{}^* Q$ either $Q \xrightarrow{\tau}$ or Q is in normal form.

As a technicality, we modify the LTS so to remove all (νr) produced by service invocations, introduce the label $r\iota$ to observe that a service invocation takes place inside session r and let ω be any sequence of τ, $r\tau$ and $r\iota$ steps.

Initial processes

- $\emptyset \vdash P : \mathrm{end}[\mathrm{end}]$
- P does not contain session constructs
- all service definitions are at the top level

Deadlock free

If P is an initial process, then it is deadlock free.

Main result

Normal form

$P \equiv (\nu s_1)\dots(\nu s_n)(s_1.Q_1|\dots|s_n.Q_n)$

Deadlock free processes

P such that whenever $P \xrightarrow{\omega}{}^{*} Q$ either $Q \xrightarrow{\tau}$ or Q is in normal form.

As a technicality, we modify the LTS so to remove all (νr) produced by service invocations, introduce the label $r\iota$ to observe that a service invocation takes place inside session r and let ω be any sequence of τ, $r\tau$ and $r\iota$ steps.

Initial processes

- $\emptyset \vdash P : \mathtt{end}[\mathtt{end}]$
- P does not contain session constructs
- all service definitions are at the top level

Deadlock free

If P is an initial process, then it is deadlock free.

Example: Factorial

Processes

$$F \equiv \quad fatt.(n).\texttt{if}\ (n = 0)$$
$$\texttt{then}\ \langle 1 \rangle$$
$$\texttt{else}\ (\overline{fatt}.\langle n-1 \rangle.(x).\texttt{return}\ x) > x > \langle n \cdot x \rangle$$

$$P \equiv \quad \overline{fatt}.\langle 3 \rangle.(x) \quad | \quad \overline{fatt}.\langle 5 \rangle.(x).\texttt{return}\ x$$

$$Q \equiv \quad P > z > \overline{fatt}.\langle z \rangle.(x)$$

Types

$$\Gamma = fatt : [?(int).!(int)], - : int \times int \to int, \cdot : int \times int \to int$$
$$\Gamma \vdash F : \mathrm{end}[\mathrm{end}]$$
$$\Gamma \vdash P : \mathrm{end}[!(int).\mathrm{end}]$$
$$\Gamma \vdash Q : \mathrm{end}[\mathrm{end}]$$
$$\emptyset \vdash (\nu fatt)(F|Q) : \mathrm{end}[\mathrm{end}]$$

Example: Factorial

Processes

$$F \equiv \quad \mathit{fatt}.(n).\texttt{if}\ (n = 0)$$
$$\texttt{then}\ \langle 1 \rangle$$
$$\texttt{else}\ (\overline{\mathit{fatt}}.\langle n - 1 \rangle.(x).\texttt{return}\ x) > x > \langle n \cdot x \rangle$$

$$P \equiv \quad \overline{\mathit{fatt}}.\langle 3 \rangle.(x) \quad | \quad \overline{\mathit{fatt}}.\langle 5 \rangle.(x).\texttt{return}\ x$$

$$Q \equiv \quad P > z > \overline{\mathit{fatt}}.\langle z \rangle.(x)$$

Types

$$\Gamma = \mathit{fatt} : [?(\mathit{int}).!(\mathit{int})], - : \mathit{int} \times \mathit{int} \to \mathit{int}, \cdot : \mathit{int} \times \mathit{int} \to \mathit{int}$$
$$\Gamma \vdash F : \mathsf{end}[\mathsf{end}]$$
$$\Gamma \vdash P : \mathsf{end}[!(\mathit{int}).\mathsf{end}]$$
$$\Gamma \vdash Q : \mathsf{end}[\mathsf{end}]$$
$$\emptyset \vdash (\nu \mathit{fatt})(F | Q) : \mathsf{end}[\mathsf{end}]$$

Outline

Conclusion and Future Work

SCC

- Original mix of several ingredients
- Flexible and expressive

Type system

- Strong result over a (reasonable) fragment of SCC
- Difficult to obtain by encoding SCC in other typed calculi

Ongoing work

- Type inference
- Subtyping
- Recursive protocols and regular session types

THANKS FOR THE ATTENTION!

Conclusion and Future Work

SCC

- Original mix of several ingredients
- Flexible and expressive

Type system

- Strong result over a (reasonable) fragment of SCC
- Difficult to obtain by encoding SCC in other typed calculi

Ongoing work

- Type inference
- Subtyping
- Recursive protocols and regular session types

THANKS FOR THE ATTENTION!