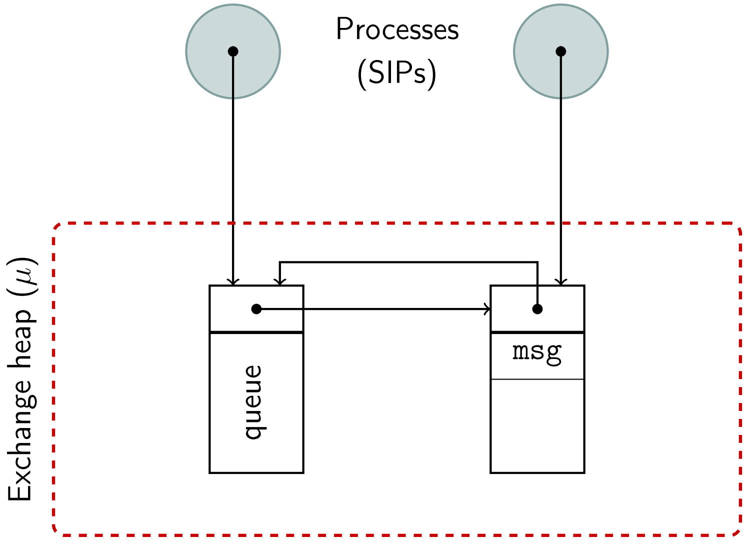
Polymorphic Endpoint Types for Copyless Message Passing

Viviana Bono   Luca Padovani

Dipartimento di Informatica, Università di Torino

ICE 2011

Singularity OS: architecture overview



Sing♯ examples

```
void CLIENT() {
  (e, f) = open();
  spawn { SERVER(f) }
  send(e, v1);
  send(e, v2);
  res = receive(e);
  close(e);
}
```

```
void SERVER(f) {
  a1 = receive(f);
  a2 = receive(f);
  ...
  send(f, OP(a1, a2));
  close(f);
}
```

Desired safety properties

1. no communication errors
2. no memory faults
3. no memory leaks

Avoiding communication errors

```
contract OP_Service {
  initial state START { Arg!<α>(α) → SEND<α> }
  state SEND<α> { Arg!(α) → WAIT }
  state WAIT { Res?bool → END }
  final state END { }
}
```

+ recursion
+ branching

Avoiding memory faults and leaks

Process isolation

- at any given time, no pointer is shared by two or more processes

Example 1

```
send(a, b);
/*** can no longer use b ***/
```

Example 2

```
send(a, *b);
/*** can use b but not *b ***/
*b = new T();
```

Enforcing safety properties

1. no communication errors
2. no memory faults
3. no memory leaks

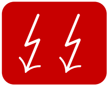
LINEAR TYPE SYSTEM

- too restrictive in some cases
- too permissive in others

Linearity is too restrictive

```
void CLIENT() {
  (e, f) = open();
  spawn { SERVER(f) }
  send(e, v1);
  send(e, v2);
  res = receive(e);
  close(e);
}
```

```
send(a, *b);
```

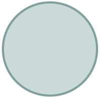


```
*b = new T();
```

- we want these

Linearity is too permissive

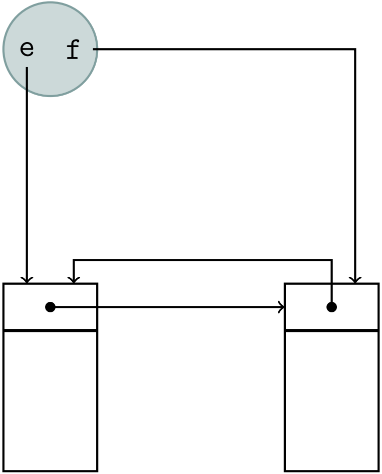
```
void FOO()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```



- we don't want this

Linearity is too permissive

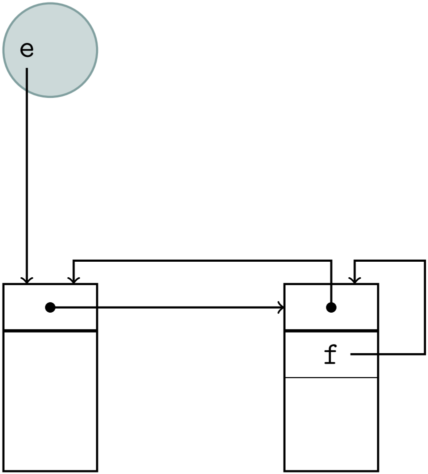
```
void FOO()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```



- we don't want this

Linearity is too permissive

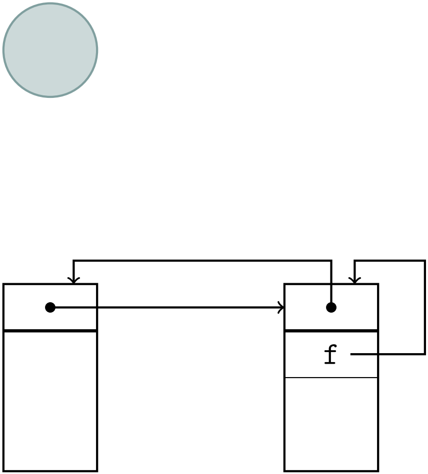
```
void FOO()
{
  (e, f) = open();
→ send(e, f);
  close(e);
}
```



- we don't want this

Linearity is too permissive

```
void FOO()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```



- we don't want this

Modeling processes

P	$::=$		**Process**
		$\mathbf{0}$	(idle)
	$\mid$	$\mathsf{open}(a, b).P$	(open channel)
	$\mid$	$\mathsf{close}(u)$	(close endpoint)
	$\mid$	$u!v.P$	(send)
	$\mid$	$u?(x).P$	(receive)
	$\mid$	$P \oplus P$	(choice)
	$\mid$	$P \mid P$	(composition)
	$\mid$	X	(variable)
	$\mid$	$\mathsf{rec}\ X.P$	(recursion)

- name = exchange heap pointer
- channel = peer endpoints
- explicit channel closure

Modeling contracts

```
contract OP_Service {
  initial state START { Arg!<α>(α) → SEND<α> }
  state SEND<α> { Arg!(α) → WAIT }
  state WAIT { Res?bool → END }
  final state END { }
}
```

Client/Import	Service/Export
$\forall\alpha.!\alpha.!\alpha.?\mathtt{bool}.\mathsf{end}$	$\exists\alpha.?\alpha.?\alpha.!\mathtt{bool}.\mathsf{end}$

Endpoint types

T	$::=$		**Endpoint Type**
		end	(termination)
	$\mid$	α	(type variable)
	$\mid$	$!\langle\alpha\rangle t.T$	(output)
	$\mid$	$?\langle\alpha\rangle t.T$	(input)
	$\mid$	X	(recursion variable)
	$\mid$	rec $X.T$	(recursive type)

Typing message passing

(T-Open)

$$\frac{\Delta, a : T, b : \overline{T} \vdash P}{\Delta \vdash \mathsf{open}(a, b).P}$$

(T-Send)

$$\frac{\Delta, u : T\{s/\alpha\} \vdash P}{\Delta, u : !\langle\alpha\rangle t.T, v : t\{s/\alpha\} \vdash u!v.P}$$

(T-Receive)

$$\frac{\alpha \text{ fresh} \qquad \Delta, u : T, x : t \vdash P}{\Delta, u : ?\langle\alpha\rangle t.T \vdash u?(x).P}$$

Typable leak

```
void foo()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```

$\mathsf{open}(\mathtt{e}, \mathtt{f}).$
$\mathtt{e}!\mathtt{f}.$
$\mathsf{close}(\mathtt{e}).$
$\mathbf{0}$

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

Typable leak

```
void foo()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```

$$\{\} \vdash \mathsf{open}(\mathtt{e}, \mathtt{f}).\ \mathtt{e}!\mathtt{f}.\ \mathsf{close}(\mathtt{e}).\ \mathbf{0}$$

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

Typable leak

```
void foo()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```

$$
\begin{array}{rl}
\{\} \vdash & \mathsf{open}(\mathtt{e}, \mathtt{f}). \\
\{\mathtt{e} : T, \mathtt{f} : \overline{T}\} \vdash & \mathtt{e}!\mathtt{f}. \\
& \mathsf{close}(\mathtt{e}). \\
& \mathbf{0}
\end{array}
$$

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

Typable leak

```
void foo()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```

$$\{\} \vdash \mathsf{open}(\mathtt{e},\mathtt{f}).$$
$$\{\mathtt{e} : T, \mathtt{f} : \overline{T}\} \vdash \mathtt{e}!\mathtt{f}.$$
$$\{\mathtt{e} : \mathsf{end}\} \vdash \mathsf{close}(\mathtt{e}).$$
$$\mathbf{0}$$

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

Typable leak

```
void foo()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```

$$\{\} \vdash \mathsf{open}(\mathtt{e}, \mathtt{f}).$$
$$\{\mathtt{e} : T, \mathtt{f} : \overline{T}\} \vdash \mathtt{e}!\mathtt{f}.$$
$$\{\mathtt{e} : \mathsf{end}\} \vdash \mathsf{close}(\mathtt{e}).$$
$$\{\} \vdash \mathbf{0}$$

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

Understanding the problem

"Improper" recursion?

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

But these are safe!

$$S = \mathsf{rec}\ X.!X.\mathsf{end} \qquad \overline{S} = ?S.\mathsf{end}$$

Understanding the problem

"Improper" recursion?

$$T = !\overline{T}.\mathsf{end} \qquad \overline{T} = \mathsf{rec}\ X.?X.\mathsf{end}$$

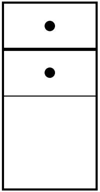
But these are safe!

$$S = \mathsf{rec}\ X.!X.\mathsf{end} \qquad \overline{S} = ?S.\mathsf{end}$$

Queue depth and self-ownership

Fact

- endpoints in "receive state" may have a non-empty queue
- "endpoint in receive state" = "endpoint has type $?t. \dots$"

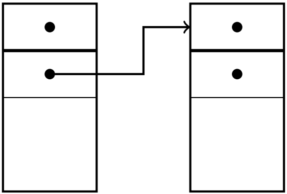


$T = ?t$

Queue depth and self-ownership

Fact

- endpoints in "receive state" may have a non-empty queue
- "endpoint in receive state" = "endpoint has type $?t.\dots$"

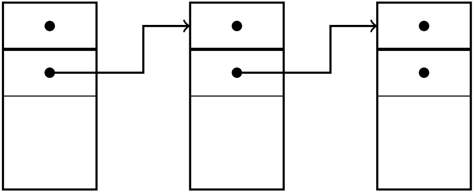


$T = ?t$ $\qquad$ $t = ?s$

Queue depth and self-ownership

Fact

- endpoints in “receive state” may have a non-empty queue
- “endpoint in receive state” = “endpoint has type $?t. \ldots$”

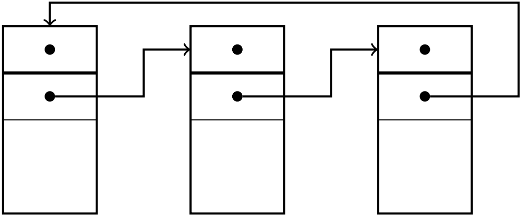


$T = ?t$ $\quad t = ?s$ $\quad s = ?T$

Queue depth and self-ownership

Fact

- endpoints in "receive state" may have a non-empty queue
- "endpoint in receive state" = "endpoint has type $?t. \dots$"



$T = ?t$ $t = ?s$ $s = ?T$

Type weight

- $\|T\|$ = "maximum length of chains of pointers from the queue of an endpoint with type T"
- only pointers whose type has finite weight can be sent

(T-Send)

$$\frac{\Delta, u : T\{s/\alpha\} \vdash P \qquad \|t\{s/\alpha\}\| < \infty}{\Delta, u : !\langle\alpha\rangle t.T, v : t\{s/\alpha\} \vdash u!v.P}$$

Type weight

- $\|T\|$ = "maximum length of chains of pointers from the queue of an endpoint with type T"
- only pointers whose type has finite weight can be sent

$$\text{(T-Send)}\ \frac{\Delta, u : T\{s/\alpha\} \vdash P \qquad \|t\{s/\alpha\}\| < \infty}{\Delta, u : !\langle\alpha\rangle t.T, v : t\{s/\alpha\} \vdash u!v.P}$$

Type weight: examples

$$T = !\overline{T}.\mathsf{end} \qquad \|T\| = 0$$

$$\overline{T} = \mathsf{rec}\ X.?X.\mathsf{end} \qquad \|\overline{T}\| = \infty$$

$$S = \mathsf{rec}\ X.!X.\mathsf{end} \qquad \|S\| = 0$$

$$\overline{S} = ?S.\mathsf{end} \qquad \|\overline{S}\| = 1$$

The weight of type variables

$$\|\alpha\| = \infty$$

$$\begin{array}{r l}
\{\} & \vdash \mathsf{open}(\mathtt{e}, \mathtt{f}). \\
\{\mathtt{e} : !\langle\alpha\rangle\alpha.\mathsf{end}, \mathtt{f} : ?\langle\alpha\rangle\alpha.\mathsf{end}\} & \vdash \mathtt{e}!\mathtt{f}. \\
\{\mathtt{e} : \mathsf{end}\} & \vdash \mathsf{close}(\mathtt{e}). \\
\{\} & \vdash \mathbf{0}
\end{array}$$

Can we do better?

Bounded polymorphism

t	$::=$		**Type**
	$\mid$	T	(endpoint type)
T	$::=$		**Endpoint Type**
		end	(termination)
	$\mid$	α	(type variable)
	$\mid$	$!\langle\alpha \quad\rangle t.T$	(output)
	$\mid$	$?\langle\alpha \quad\rangle t.T$	(input)
	$\mid$	X	(recursion variable)
	$\mid$	$\mathsf{rec}\ X.T$	(recursive type)

Bounded polymorphism

- S. Gay, **Bounded Polymorphism in Session Types**, 2008

t	$::=$		**Type**
		$\mathtt{Top}$	(top type)
	$\mid$	T	(endpoint type)

T	$::=$		**Endpoint Type**
		end	(termination)
	$\mid$	α	(type variable)
	$\mid$	$!\langle\alpha \leqslant s\rangle t.T$	(output)
	$\mid$	$?\langle\alpha \leqslant s\rangle t.T$	(input)
	$\mid$	X	(recursion variable)
	$\mid$	$\mathsf{rec}\ X.T$	(recursive type)

On the weight of type variables

Proposition

If $t \leqslant s$, then $\|t\| \leq \|s\|$.

- α has a type bound $\alpha \leqslant t$
- α is always instantiated with some $s \leqslant t$
- $\|\alpha\|$ has weight bound $\|t\|$

Examples

- $\|?\langle\alpha\rangle\alpha.\mathsf{end}\| = \infty$
- $\|?\langle\alpha \leqslant t\rangle\alpha.\mathsf{end}\| < \infty$ if t has finite weight

Well-behaved processes

P is well behaved if $(\emptyset; P) \Rightarrow (\mu; Q)$ implies:

1. $\mathrm{reach}(\mathrm{fn}(Q), \mu) \subseteq \mathrm{dom}(\mu)$
2. $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q), \mu)$
3. $Q \equiv P_1 \mid P_2$ implies $\mathrm{reach}(\mathrm{fn}(P_1), \mu) \cap \mathrm{reach}(\mathrm{fn}(P_2), \mu) = \emptyset$
4. $Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \not\rightarrow$ where P_1 does not have unguarded parallel compositions imply either
   - $P_1 = \mathbf{0}$, or
   - $P_1 = a?(x).P$ where the queue of a is empty

Well-behaved processes

P is well behaved if $(\emptyset; P) \Rightarrow (\mu; Q)$ implies:

1. $\mathrm{reach}(\mathrm{fn}(Q), \mu) \subseteq \mathrm{dom}(\mu)$
2. $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q), \mu)$
3. $Q \equiv P_1 \mid P_2$ implies $\mathrm{reach}(\mathrm{fn}(P_1), \mu) \cap \mathrm{reach}(\mathrm{fn}(P_2), \mu) = \emptyset$
4. $Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \not\rightarrow$ where P_1 does not have unguarded parallel compositions imply either
   - $P_1 = \mathbf{0}$, or
   - $P_1 = a?(x).P$ where the queue of a is empty

Well-behaved processes

P is well behaved if $(\emptyset; P) \Rightarrow (\mu; Q)$ implies:

1. $\mathrm{reach}(\mathrm{fn}(Q), \mu) \subseteq \mathrm{dom}(\mu)$
2. $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q), \mu)$
3. $Q \equiv P_1 \mid P_2$ implies $\mathrm{reach}(\mathrm{fn}(P_1), \mu) \cap \mathrm{reach}(\mathrm{fn}(P_2), \mu) = \emptyset$
4. $Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \not\rightarrow$ where P_1 does not have unguarded parallel compositions imply either
   - $P_1 = \mathbf{0}$, or
   - $P_1 = a?(x).P$ where the queue of a is empty

Well-behaved processes

P is well behaved if $(\emptyset; P) \Rightarrow (\mu; Q)$ implies:

1. $\mathrm{reach}(\mathrm{fn}(Q), \mu) \subseteq \mathrm{dom}(\mu)$
2. $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q), \mu)$
3. $Q \equiv P_1 \mid P_2$ implies $\mathrm{reach}(\mathrm{fn}(P_1), \mu) \cap \mathrm{reach}(\mathrm{fn}(P_2), \mu) = \emptyset$
4. $Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \not\rightarrow$ where P_1 does not have unguarded parallel compositions imply either
    - $P_1 = \mathbf{0}$, or
    - $P_1 = a?(x).P$ where the queue of a is empty

Results

Theorem (Subject reduction)

If $\Delta \vdash P$ and $(\mu; P) \to (\mu'; P')$, then $\Delta' \vdash P'$ for some Δ'.

Theorem (Soundness)

If $\vdash P$, then P is well behaved.

Concluding remarks

Formalization of Sing♯

- contracts $\Rightarrow$ endpoint types (= session types)
- first formalization of polymorphic Sing♯ contracts
- finite-weight restriction on type of messages
  (weight $\neq$ bound of queues)

Sing♯ restrictions

- Sing♯ forbids sending endpoints in "receive state"...
- ...for implementative reasons
- Sing♯ is leak-free, incidentally? ☺

Related work

- Bono, Messa, Padovani, **Typing Copyless Message Passing**, ESOP 2011 (no polymorphism)

A different approach based on separation logic

- Villard, Lozes, Calcagno, **Proving Copyless Message Passing**, APLAS 2009
- Villard, Lozes, Calcagno, **Tracking heaps that hop with heap-hop**, TACAS 2010
- Villard, **Heaps and Hops**, PhD Thesis, 2011

Ongoing work

- subtyping algorithm
- non-linear values