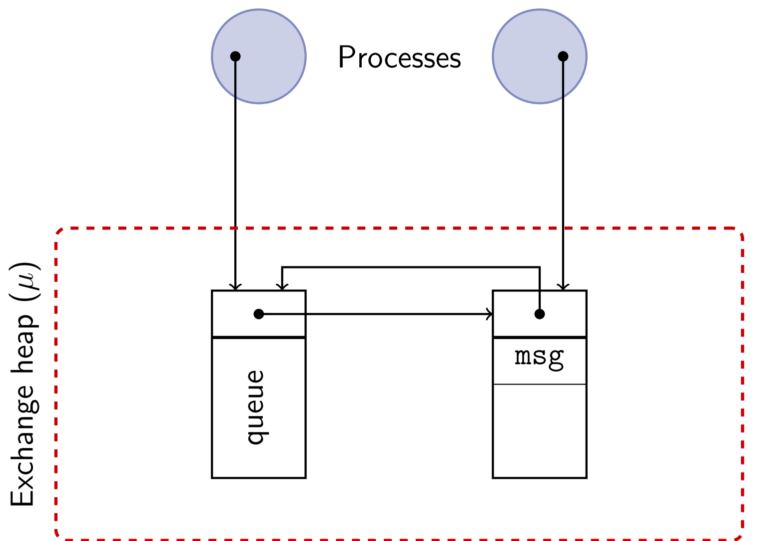
Typing Copyless Message Passing

Viviana Bono Chiara Messa Luca Padovani

Dipartimento di Informatica, Università di Torino

ESOP 2011

Singularity OS: architecture



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);
}
```

Safety properties

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

Contracts

```
contract OP_Service {
  initial state START { Arg! → WAIT_ARG_2 }
  state WAIT_ARG_2 { Arg! → WAIT_RES }
  state WAIT_RES { Res? → END }
  final state END { }
}
```

+ recursion
+ branching

Exposing structures

```
expose (a) {
  send(*a, b);
}
```

```
expose (b) {
  send(a, *b);
  *b = new T();
}
```

+ records with named fields (not in the paper)

Enforcing safety properties

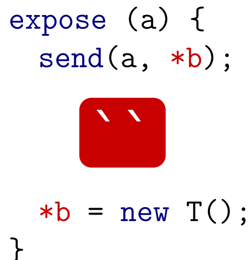
1. no communication errors
2. no memory faults
3. no memory leaks

LINEAR TYPE SYSTEM!

- too restrictive in some cases
- too permissive in others

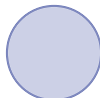
Linearity is too restrictive

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



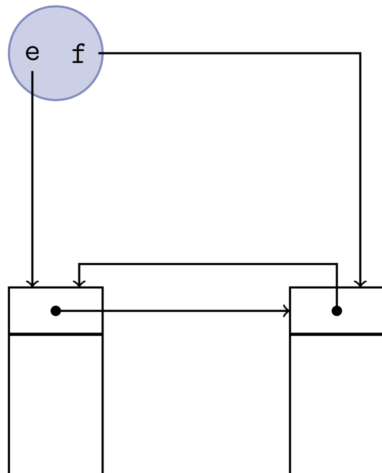
Linearity is too permissive

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



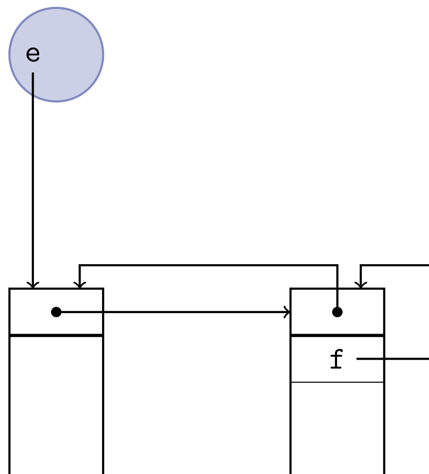
Linearity is too permissive

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

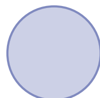


Linearity is too permissive

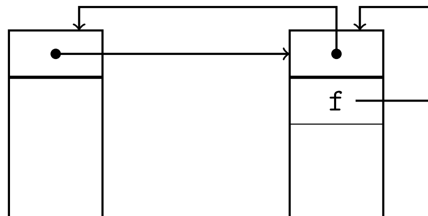
```
void foo()
{
  (e, f) = open();
  send(e, f);
  close(e);
}
```



Linearity is too permissive



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



Modeling processes

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

```
open(e, f).(SERVER |
  e!v1.
  e!v2.
  e?(res).
  free(e).
  0
)
```

- channel = peer endpoints
- explicit channel closure

Modeling exposures

```
expose (a) {
  send(*a, b);
}
```

$\mathsf{expose}(\mathtt{a}, x).$
$x!\mathtt{b}.$
$\mathsf{unexpose}(\mathtt{a}, x).\ \ \ldots$

```
expose (b) {
  send(a, *b);
  *b = new T();
}
```

$\mathsf{expose}(\mathtt{b}, x).$
$\mathtt{a}!x.$
$\mathsf{cell}(c).$
$\mathsf{unexpose}(\mathtt{b}, c).\ \ \ldots$

- expose/unexpose $\sim$ dereferentiation/assignment
- with type effects

Modeling contracts

```
contract OP_Service {
  initial state START { Arg! → WAIT_ARG_2 }
  state WAIT_ARG_2 { Arg! → WAIT_RES }
  state WAIT_RES { Res? → END }
  final state END { }
}
```

Client/Import	Service/Export
!Arg.!Arg.?Res.end	?Arg.?Arg.!Res.end

Types and endpoint types

t	$::=$		**Type**
		$*t$	(cell type)
	$\mid$	$*\bullet$	(exposed cell type)
	$\mid$	T	(endpoint type)

T	$::=$		**Endpoint Type**
		end	(termination)
	$\mid$	X	(variable)
	$\mid$	$!t.T$	(output)
	$\mid$	$?t.T$	(input)
	$\mid$	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)

$$\frac{\Delta, u : T \vdash P}{\Delta, u : !t.T, v : t \vdash u!v.P}$$

(T-Receive)

$$\frac{\Delta, u : T, x : t \vdash P}{\Delta, u : ?t.T \vdash u?(x).P}$$

Typing exposures

(T-Expose)

$$\frac{\Delta, u : *\bullet, x : t \vdash P}{\Delta, u : *t \vdash \mathsf{expose}(u, x).P}$$

(T-Unexpose)

$$\frac{\Delta, u : *t \vdash P}{\Delta, u : *\bullet, v : t \vdash \mathsf{unexpose}(u, v).P}$$

Typing exposures: example

$\mathsf{expose}(\mathtt{a}, x).$

$x!\mathtt{b}.$

$\mathsf{unexpose}(\mathtt{a}, x).$

$\ldots$

Typing exposures: example

$$\{\mathtt{a} : *(!s.T), \mathtt{b} : s\} \vdash \mathsf{expose}(\mathtt{a}, x).$$

$$x!\mathtt{b}.$$

$$\mathsf{unexpose}(\mathtt{a}, x).$$

$$\ldots$$

Typing exposures: example

$$\{\mathtt{a} : *(!s.T), \mathtt{b} : s\} \vdash \text{expose}(\mathtt{a}, x).$$

$$\{\mathtt{a} : *\bullet, x :!s.T, \mathtt{b} : s\} \vdash x!\mathtt{b}.$$

$$\text{unexpose}(\mathtt{a}, x).$$

$$\dots$$

Typing exposures: example

$$\{\mathtt{a} : *(!s.T), \mathtt{b} : s\} \vdash \text{expose}(\mathtt{a}, x).$$

$$\{\mathtt{a} : *\bullet, x : !s.T, \mathtt{b} : s\} \vdash x!\mathtt{b}.$$

$$\{\mathtt{a} : *\bullet, x : T\} \vdash \text{unexpose}(\mathtt{a}, x).$$

$$\ldots$$

Typing exposures: example

$$\{\mathtt{a} : *(!s.T), \mathtt{b} : s\} \vdash \mathsf{expose}(\mathtt{a}, x).$$

$$\{\mathtt{a} : *\bullet, x : !s.T, \mathtt{b} : s\} \vdash x!\mathtt{b}.$$

$$\{\mathtt{a} : *\bullet, x : T\} \vdash \mathsf{unexpose}(\mathtt{a}, x).$$

$$\{\mathtt{a} : *T\} \vdash \ldots$$

Typable leak

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

open(e, f).
e!f.
free(e).
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);
}
```

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

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

Typable leak

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

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

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

Typable leak

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

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

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

Typable leak

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

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

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

Understanding the problem

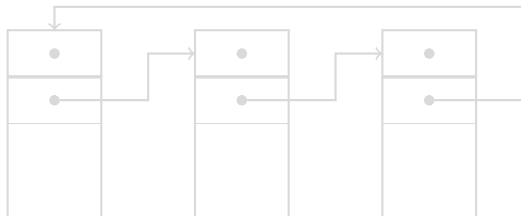
"Improper" recursion?

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

No, the following endpoint types are safe

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

It's a matter of "ownership"



Understanding the problem

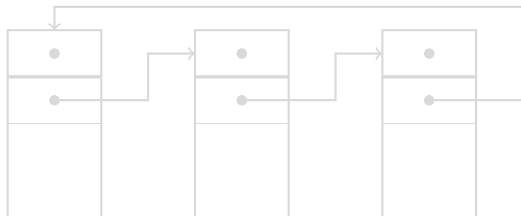
"Improper" recursion?

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

No, the following endpoint types are safe

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

It's a matter of "ownership"



Understanding the problem

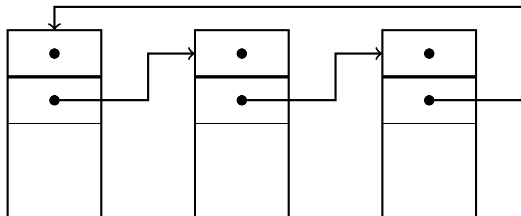
"Improper" recursion?

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

No, the following endpoint types are safe

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

It's a matter of "ownership"



Type weight

In summary

- "receive state" = "has type $?T.S$"
- only endpoints in "receive state" can have a non-empty queue

Solution

- $\|T\|$ = "depth of the queue of an endpoint with type T"
- only endpoint types with finite weight are admitted

$$\begin{aligned} T &= !\overline{T}.\mathsf{end} \\ \|T\| &= 0 \end{aligned} \qquad \begin{aligned} \overline{T} &= \mathsf{rec}\ X.?X.\mathsf{end} \\ \|\overline{T}\| &= \infty \end{aligned}$$

$$\begin{aligned} S &= \mathsf{rec}\ X.!X.\mathsf{end} \\ \|S\| &= 0 \end{aligned} \qquad \begin{aligned} \overline{S} &= ?S.\mathsf{end} \\ \|\overline{S}\| &= 1 \end{aligned}$$

Type weight

In summary

- "receive state" = "has type $?T.S$"
- only endpoints in "receive state" can have a non-empty queue

Solution

- $\|T\|$ = "depth of the queue of an endpoint with type T"
- only endpoint types with finite weight are admitted

$$T = !\overline{T}.\text{end} \qquad \overline{T} = \text{rec}\ X.?X.\text{end}$$
$$\|T\| = 0 \qquad \|\overline{T}\| = \infty$$

$$S = \text{rec}\ X.!X.\text{end} \qquad \overline{S} = ?S.\text{end}$$
$$\|S\| = 0 \qquad \|\overline{S}\| = 1$$

Type weight

In summary

- "receive state" $=$ "has type $?T.S$"
- only endpoints in "receive state" can have a non-empty queue

Solution

- $\|T\| =$ "depth of the queue of an endpoint with type T"
- only endpoint types with finite weight are admitted

$$
\begin{array}{rcl}
T & = & !\overline{T}.\mathsf{end} \\
\|T\| & = & 0
\end{array}
\qquad
\begin{array}{rcl}
\overline{T} & = & \mathsf{rec}\ X.?X.\mathsf{end} \\
\|\overline{T}\| & = & \infty
\end{array}
$$

$$
\begin{array}{rcl}
S & = & \mathsf{rec}\ X.!X.\mathsf{end} \\
\|S\| & = & 0
\end{array}
\qquad
\begin{array}{rcl}
\overline{S} & = & ?S.\mathsf{end} \\
\|\overline{S}\| & = & 1
\end{array}
$$

On weights and reachability

Proposition

If $a : T, b : S$ and $b \in \mathrm{reach}(a, \mu)$, then $\|S\| < \|T\|$.

Finite weight $\neq$ bounded queue

$$T = \text{rec } X.?\mathtt{int}.X \qquad \|T\| = 1$$

Finite weight $\neq$ acyclic heap

$$*(?*\bullet.\text{end})$$

Well-behaved processes

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

1. $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\mu)$
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. $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\mu)$
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. $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\mu)$
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. $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\mu)$
2. $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q), \mu)$
3. ~~$Q \equiv P_1 \mid P_2$ implies $\mathrm{reach}(\mathrm{fn}(P_1), \mu) \cap \mathrm{reach}(\mathrm{fn}(P_2), \mu) = \emptyset$~~
   $Q \equiv P_1 \mid P_2$ implies $\mathrm{fn}(P_1) \cap \mathrm{fn}(P_2) = \emptyset$
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. $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\mu)$
2. $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q), \mu)$
3. ~~$Q \equiv P_1 \mid P_2$ implies $\mathrm{reach}(\mathrm{fn}(P_1), \mu) \cap \mathrm{reach}(\mathrm{fn}(P_2), \mu) = \emptyset$~~
   $Q \equiv P_1 \mid P_2$ implies $\mathrm{fn}(P_1) \cap \mathrm{fn}(P_2) = \emptyset$
4. $Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \nrightarrow$ 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* $(\mu; P) \rightarrow (\mu'; P')$, *then* $\Delta' \vdash P'$ *for some* Δ'.

Theorem (Soundness)

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

Concluding remarks

Formalization of Sing#

- contracts ⇒ endpoint types (= session types)
- `expose` ⇒ opaque references $*\bullet$ (= simple behavioral types)

Sing# restrictions

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

What next: polymorphic endpoint types

Modeling parametric contracts

$$!\langle\alpha\rangle(\alpha).?(\alpha).\mathsf{end}$$

But...

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

$$\mathtt{e} : !\langle\alpha\rangle(\alpha).\mathsf{end} \qquad \mathtt{f} : ?\langle\alpha\rangle(\alpha).\mathsf{end}$$

Idea: bounded polymorphism

- $!\langle\alpha \leqslant T\rangle(\alpha).\mathsf{end}$
- $T \leqslant S$ implies $\|T\| \leq \|S\|$

What next: polymorphic endpoint types

Modeling parametric contracts

$$!\langle\alpha\rangle(\alpha).?(\alpha).\mathsf{end}$$

But...

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

$$\mathtt{e} : !\langle\alpha\rangle(\alpha).\mathsf{end} \qquad \mathtt{f} : ?\langle\alpha\rangle(\alpha).\mathsf{end}$$

Idea: bounded polymorphism

- $!\langle\alpha \leqslant T\rangle(\alpha).\mathsf{end}$
- $T \leqslant S$ implies $\|T\| \leq \|S\|$

What next: polymorphic endpoint types

Modeling parametric contracts

$$!\langle\alpha\rangle(\alpha).?(\alpha).\mathsf{end}$$

But...

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

$$\mathtt{e} : !\langle\alpha\rangle(\alpha).\mathsf{end} \qquad \mathtt{f} : ?\langle\alpha\rangle(\alpha).\mathsf{end}$$

Idea: bounded polymorphism

- $!\langle\alpha \leqslant T\rangle(\alpha).\mathsf{end}$
- $T \leqslant S$ implies $\|T\| \leq \|S\|$

Thank you