Proving Copyless Message Passing

Jules Villard[1] Étienne Lozes[1] Cristiano Calcagno[2]

[1]LSV, ENS Cachan, CNRS

[2]Imperial College, London

ANR PANDA – Sept. 10 – PPS

Outline

Inspiration: Singularity [Fähndrich & al. '06]

Singularity: a research project and an operating system.

- No memory protection: all processes share the same address space
- Memory isolation is verified at compile time (Sing♯ language)
- No shared resources. Instead, processes communicate by copyless message passing
- Communications are ruled by contracts
- Many guarantees ensured by the compiler:
  - race freedom (process isolation)
  - contract obedience
  - progress (?)

Sing♯ communication model

- Channels are bidirectional and asynchronous

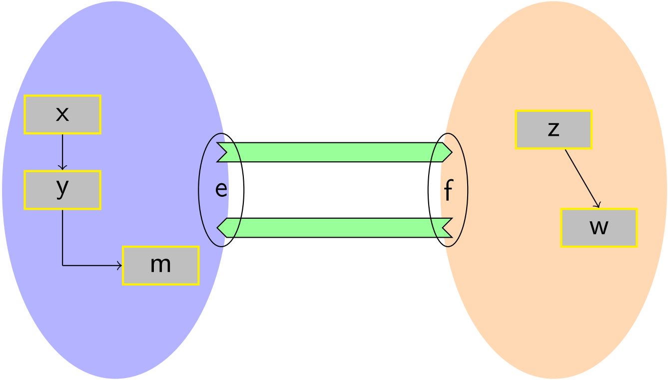
  channel = pair of FIFO queues
- Channels are made of two endpoints

  similar to socket model
- Endpoints are allocated, disposed of, and may be communicated through channels

  under some conditions, similar to internal mobility in π-calculus
- Communications are ruled by user-defined contracts

  similar to session types

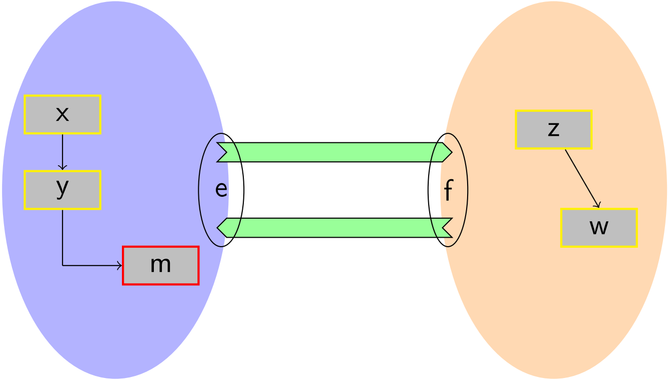
Message Passing with copies



`send(cell,e,m);`

`*z = receive(cell,f);`

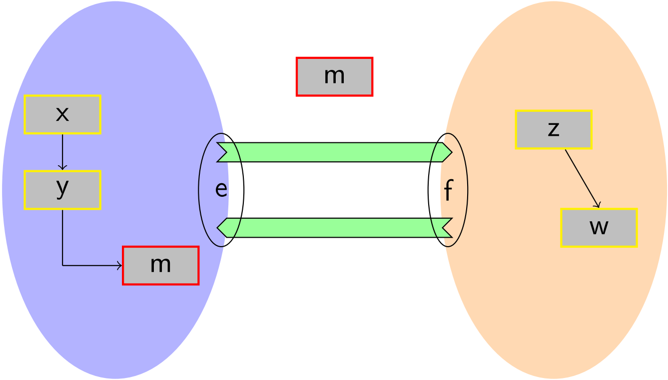
Message Passing with copies




```
send(cell,e,m);
```

```
*z = receive(cell,f);
```

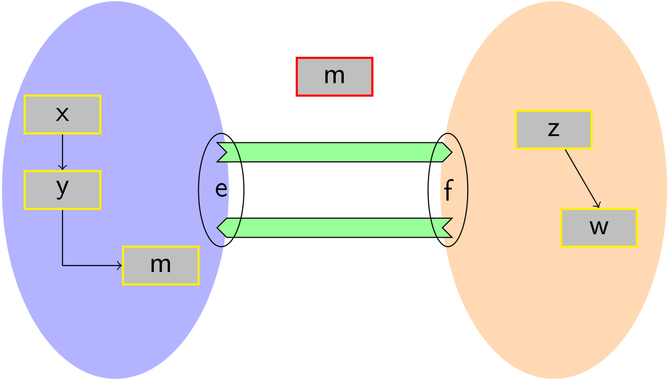
Message Passing with copies




```
send(cell,e,m);
```

```
*z = receive(cell,f);
```

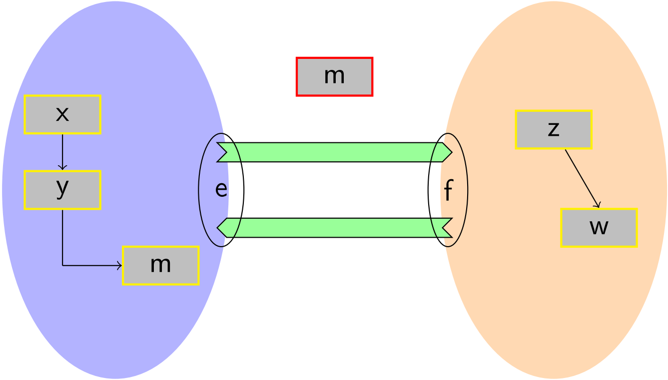
Message Passing with copies




```
send(cell,e,m);
```

```
*z = receive(cell,f);
```

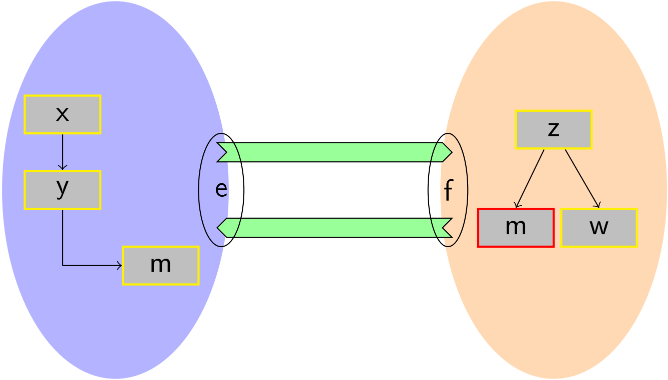
Message Passing with copies




```
send(cell,e,m);
```

```
*z = receive(cell,f);
```

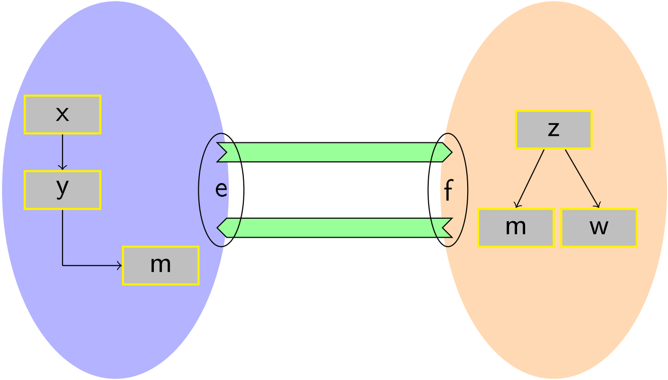
Message Passing with copies



`send(cell,e,m);`

`*z = receive(cell,f);`

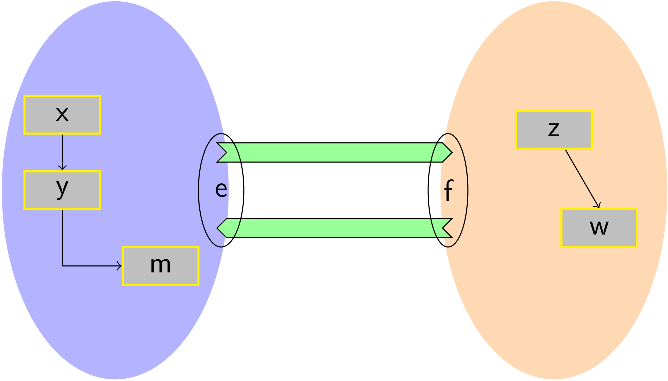
Message Passing with copies




```
send(cell,e,m);
```

```
*z = receive(cell,f);
```

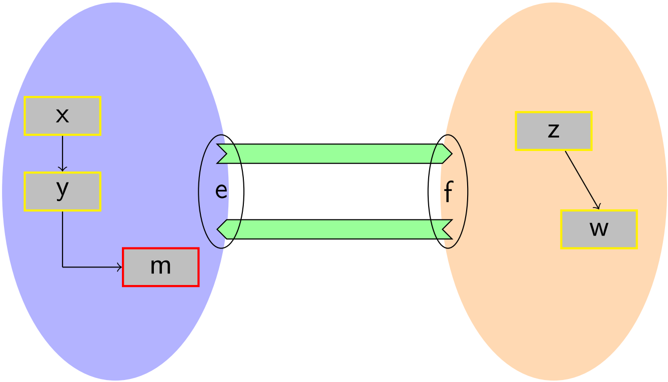
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`*z = receive(cell,f);`

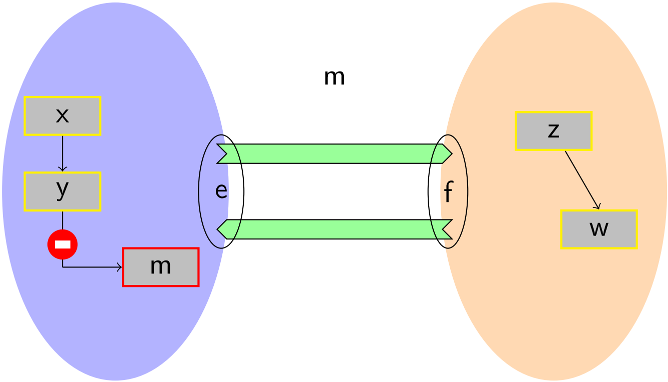
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`*z = receive(cell,f);`

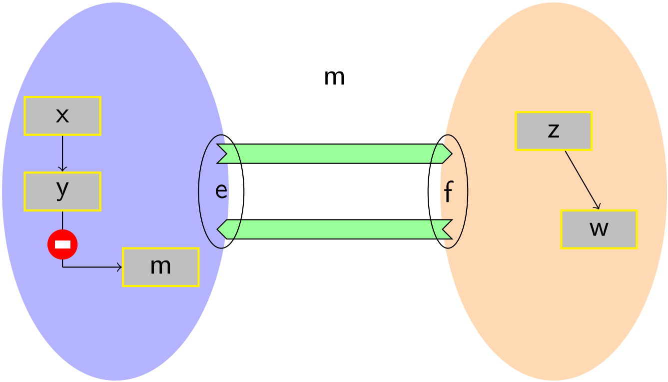
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`*z = receive(cell,f);`

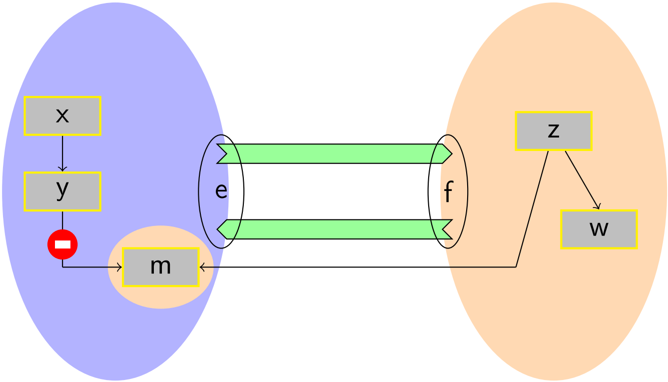
Copyless Message Passing (shared memory)




```
send(cell,e,m);
```

```
*z = receive(cell,f);
```

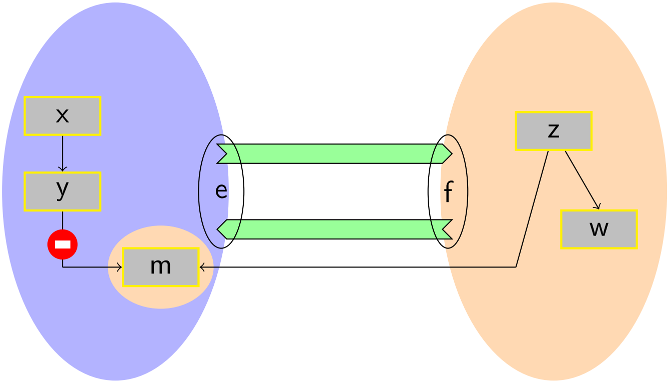
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`*z = receive(cell,f);`

Copyless Message Passing (shared memory)



`send(cell,e,m);`

`*z = receive(cell,f);`

In this talk [APLAS'09]

- Define a simple model of this language
- Provide a proof system based on Separation Logic

In this talk [APLAS'09]

- Define a simple model of this language
- Provide a proof system based on Separation Logic
  - Validate programs w.r.t. ownership
  - Compositional approach
  - Provide a tool for annotated programs

Syntax of the Programming Language

Expressions and Boolean Expressions

$$E ::= x \in Var \mid \ell \in Loc \mid \varepsilon \in Endpoint \mid v \in Val$$
$$B ::= E = E \mid B \text{ and } B \mid \text{not } B$$

Atomic commands

$$c ::= \mathsf{x = E}$$
$$\mid \mathsf{x} = \mathsf{new}() \mid \mathsf{dispose}(\mathsf{x}) \mid \mathsf{x = E{\rightarrow}f} \mid \mathsf{x{\rightarrow}f = E} \mid \ldots$$

Programs

$$p ::= c \mid p; p \mid p||p \mid \text{if } B \text{ then } p \text{ else } p \mid \text{while } B\ \{p\} \mid \text{local } \mathsf{x} \text{ in } p$$

Syntax of atomic commands (continued)

c ::=	...	
	| (e, f) = open(C)	(creates a channel with endpoints e,f)
	| close(E,E')	(channel disposal)
	| send(m, E, E')	(sends message m over endpoint E)
	| x = receive(m, E)	(receives message m over endpoint E)

Comments

- m is a message identifier, not the value of the message
- both endpoints of a channel must be closed together

```
local e,f in
  (e,f) = open(C);
  send(m,e,a);
  b = receive(m,f);
  close(e,f);
```

$\approx$

```
b = a;
```

Channels, Contracts

Processes communicate through channels.

- A channel is made of two endpoints.
- It is bidirectional and asynchronous.
- It must follow a contract.

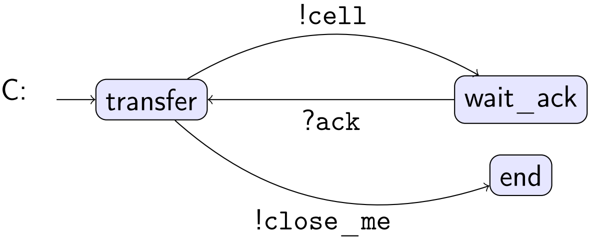
Contracts dictate which sequences of messages are admissible.

- It is a finite state machine, where arrows are labeled by a message's name and a direction: send (!) or receive (?).
- Dual endpoints of a channel follow dual contracts ($\bar{C} = C[? \leftrightarrow !]$).
- We consider leak-free contracts that ensure absence of memory leaks

Contract Example

```
message ack
message cell
message close_me

contract C {
  initial state transfer { !cell -> wait;
                           !close_me -> end; }
  state wait { ?ack -> transfer; }
  final state end {}
}
```



HEAP HOP

heaps that hop!

Outline

Separation Logic [O'Hearn 01, Reynolds 02, . . .]

- An assertion language to describe states
- An extension of Hoare Logic

Syntax

$$
\begin{array}{llll}
E & ::= & x \mid n \in \mathbb{N} & \text{expressions} \\
A & ::= & E_1 = E_2 \mid E_1 \neq E_2 & \text{stack predicates} \\
 & & \mid \mathsf{emp_h} \mid E_1 \mapsto E_2 & \text{heap predicates} \\
 & & \mid A_1 \wedge A_2 \mid A_1 * A_2 & \text{formulas}
\end{array}
$$

Semantics

$$
\begin{array}{llll}
(s,h) \vDash & E_1 = E_2 & \text{iff} & [\![E_1]\!]s = [\![E_2]\!]s \\
(s,h) \vDash & \mathsf{emp}_h & \text{iff} & dom(h) = \emptyset \\
(s,h) \vDash & E_1 \mapsto E_2 & \text{iff} & dom(h) = \{[\![E_1]\!]s\} \ \& \ h([\![E_1]\!]s) = [\![E_2]\!]s \\
(s,h) \vDash & A_1 \wedge A_2 & \text{iff} & (s,h) \vDash A_1 \ \& \ (s,h) \vDash A_2 \\
(s,h) \vDash & A_1 * A_2 & \text{iff} & \exists h_1, h_2.\, dom(h_1) \cap dom(h_2) = \emptyset \\
 & & & \qquad \& \ h = h_1 \cup h_2 \\
 & & & \qquad \& \ (s,h_1) \vDash A_1 \ \& \ (s,h_2) \vDash A_2
\end{array}
$$

Assertion Language (extension)

Syntax (continued)

$$A \quad ::= \quad \ldots$$
$$\mid \mathsf{emp}_{\mathsf{ep}} \mid E \overset{peer}{\mapsto} (C\{a\}, E') \qquad \text{endpoints' predicates}$$

Intuitively $E \overset{peer}{\mapsto} (C\{a\}, E')$ means :

- E is an allocated endpoint
- its peer is E'
- it is ruled by contract C
- it currently is in contract's state a

True/False

1. $x \mapsto d : 10 * y \mapsto d : 11$ — satisfiable, 2 cells
2. $x \mapsto d : 10 \wedge y \mapsto d : 11$ — false
3. $x \mapsto d : 10 \wedge y \mapsto d : 10$ — satisfiable, $x = y$
4. $x \mapsto - \wedge x \overset{peer}{\mapsto} (-, -)$ — false

Theorem 1 (Soundness)

If a Hoare triple $\{A\}\ p\ \{B\}$ is provable, then if the program p starts in a state satisfying A and terminates,

1. *p does not fault on memory accesses*
2. *p does not leak memory*
3. *the final state satisfies B*

Standard Hoare Logic

$$\frac{\{A\}\ p\ \{A'\} \qquad \{A'\}\ p'\ \{B\}}{\{A\}\ p; p'\ \{B\}} \qquad \ldots$$

Local Reasoning Rules

$$\frac{\{A\}\ p\ \{B\}}{\{A * F\}\ p\ \{B * F\}} \qquad \frac{\{A\}\ p\ \{B\} \qquad \{A'\}\ p'\ \{B'\}}{\{A * A'\}\ p || p'\ \{B * B'\}}$$

Small Axioms

$$\{A\}\ \mathsf{x}\ =\ \mathsf{E}\ \{A[x \leftarrow x'] \wedge x = E[x \leftarrow x']\}$$

$$\{\mathsf{emp}\}\ \mathsf{x}\ =\ \mathsf{new}()\ \{\exists v. x \mapsto v\} \qquad \ldots$$

$\{\ x \mapsto d : 10\ \}$

```
y = new();
```

$\{\ x \mapsto d : 10 * y \mapsto -\ \}$

```
y->d = 42;
```

$\{\ x \mapsto d : 10 * y \mapsto d : 42\ \}$

```
dispose(x);
```

$\{\ y \mapsto d : 42\ \}$

```
x = y;
```

$\{\ x \mapsto d : 42 \wedge x = y\ \}$

Proof System (extended)

Standard Hoare Logic

Unchanged.

Local Reasoning Rules

Unchanged.

Small Axioms

Small axioms added for new commands.

Annotating Messages

- We have to know the contents of messages
- Each message m appearing in a contract is described by a formula I_m of our logic.

- I_m may refer to two special variables:
  - `val` will denote the location of the message in memory
  - `src` will denote the location of the sending endpoint

Small Axioms for Communications

Receive rule:

$$\frac{a \xrightarrow{?m} b \in C}{\{E \overset{peer}{\mapsto} (C\{a\}, f)\}\ \mathtt{x = receive(m, E)}\ \{E \overset{peer}{\mapsto} (C\{b\}, f) * I_m(x, f)\}}$$

Send rules:

$$\frac{a \xrightarrow{!m} b \in C}{\{E \overset{peer}{\mapsto} (C\{a\}, -) * I_m(E', E)\}\ \textsf{send(E.m,E')}\ \{E \overset{peer}{\mapsto} (C\{b\}, -)\}}$$

$$\frac{a \xrightarrow{!m} b \in C}{\begin{array}{c}\{E \overset{peer}{\mapsto} (C\{a\}, -) * (E \overset{peer}{\mapsto} (C\{b\}, -) \mathrel{-\!\!*} I_m(E', E))\} \\ \textsf{send(E.m,E')} \\ \{\textsf{emp}\}\end{array}}$$

Small Axioms for Communications

Open and Close rules:

$$\frac{i = \mathrm{init}(C)}{\{\mathsf{emp}\}\ (\mathsf{e},\mathsf{f})\ =\ \mathsf{open}(\mathsf{C})\ \{e \overset{peer}{\mapsto} (C\{i\}, f) * f \overset{peer}{\mapsto} (\bar{C}\{i\}, e)\}}$$

$$\frac{f \in \mathrm{final}(C)}{\{E \overset{peer}{\mapsto} (C\{f\}, E') * E' \overset{peer}{\mapsto} (\bar{C}\{f\}, E)\}\ \mathsf{close}(\mathsf{E},\mathsf{E'})\ \{\mathsf{emp}\}}$$

- Why is the `close` rule sound?

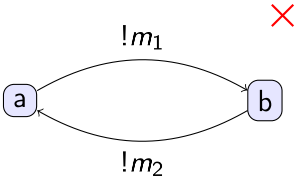
$$\frac{f \in \mathrm{final}(C)}{\{E \overset{peer}{\mapsto} (C\{f\}, E') * E' \overset{peer}{\mapsto} (\bar{C}\{f\}, E)\}\ \mathsf{close}(E,E')\ \{\mathsf{emp}\}}$$

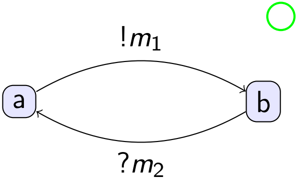
Leak-free Contracts

A contract $\mathcal{C}$ is leak-free if whenever both ends of a channel ruled by $\mathcal{C}$ are in the same final state, there are no pending messages in the channel.

Definition 2 (Synchronizing state)

A state s is synchronizing if every cycle that goes through it contains at least one send and one receive.



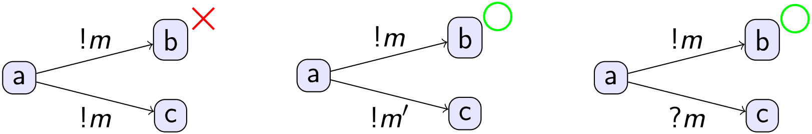


Properties of Contracts

Definition 2 (Synchronizing state)

Definition 3 (Determinism)

Two distinct edges in a contract must be labeled by different messages.

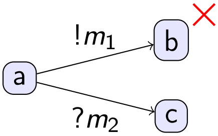


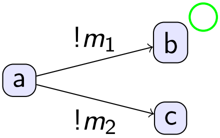
Definition 2 (Synchronizing state)

Definition 3 (Determinism)

Definition 4 (Uniform choice)

All outgoing edges from a same state in a contract must be either all sends or all receives.





Properties of Contracts

Definition 2 (Synchronizing state)

Definition 3 (Determinism)

Definition 4 (Uniform choice)

Lemma 5 (Half-Duplex)

3 & 4 $\Rightarrow$ *communications are half-duplex.*

Lemma 6 (Leak-free)

final states are synchronizing and communications are half-duplex $\Rightarrow$ *contract is leak-free*

Theorem 7 (Soundness for Copyless Message Passing)

If a Hoare triple $\{A\}\ p\ \{B\}$ is provable and the contracts are leak free, then if the program p starts in a state satisfying A and terminates,

1. *contracts are respected*
2. *p does not fault on memory accesses*
3. *p does not leak memory*
4. *the final state satisfies B*
5. *there is no race*
6. *no communication error occur*
7. *there is no deadlock*

Theorem 7 (Soundness for Copyless Message Passing)

If a Hoare triple $\{A\}\ p\ \{B\}$ is provable and the contracts are leak free, then if the program p starts in a state satisfying A and terminates,

1. *contracts are respected*
2. *p does not fault on memory accesses*
3. *p does not leak memory* — *thanks to contracts!*
4. *the final state satisfies B*
5. *there is no race*
6. *no communication error occur* — *thanks to contracts!*
7. *there is no deadlock*

Theorem 7 (Soundness for Copyless Message Passing)

If a Hoare triple $\{A\}\ p\ \{B\}$ is provable and the contracts are leak free, then if the program p starts in a state satisfying A and terminates,

1. *contracts are respected*
2. *p does not fault on memory accesses*
3. *p does not leak memory* — *thanks to contracts!*
4. *the final state satisfies B*
5. *there is no race*
6. *no communication error occur* — *thanks to contracts!*
7. *there is no deadlock* — *not yet...*

Proof of the Example

```
                //list(x)
                local e,f;
                (e,f) = open(C);
//list(x) * e|->(C{i},f) * f|->(C{i},e)
//(list(x)*e|->(C{i},f)) * (f|->(C{i},e))
```

```
local t;                          local y, e=0;
while (x != null) {               while (e == 0) {
  t = x->tl;                        { y = receive(cell, f);
  send(cell, e, x);                   free(y);
  x = t;                              send(ack, f);
  receive(ack, e); }          ||    } + {
send(close_me, e, e);                 e = receive(close_me, f);
                                    }}
                                  close(e, f);
```

```
// list(x) * e|->(C{i},f)
local t;
while (x != null) {

  t = x->tl;

  send(cell, e, x);

  x = t;
  receive(ack, e); }

send(close_me, e, e);
```

Proof of the Example

```
// list(x) * e|->(C{i},f)
local t;
while (x != null) {
  // x|-> Y * ls(Y) * e|->(C{i},f)
  t = x->tl;
  // x|-> Y * ls(Y) * e|->(C{i},f) /| t=Y
  send(cell, e, x);
  // list(t) * e|->(C{ack},f)
  x = t;
  receive(ack, e); }
// e|->(C{transfer},f)
send(close_me, e, e);
// emp
```

Proof of the Example

```
              //list(x)
              local e,f;
              (e,f) = open(C);
//list(x) * e|->(C{i},f) * f|->(C{i},e)
//(list(x)*e|->(C{i},f)) * (f|->(C{i},e))
```

```
local t;                        local y, e=0;
while (x != null) {             while (e == 0) {
  t = x->tl;                      { y = receive(cell, f);
  send(cell, e, x);                 free(y);
  x = t;                            send(ack, f);
  receive(ack, e); }    ||        } + {
send(close_me, e, e);               e = receive(close_me, f);
                                  }}
                                close(e, f);
```

Proof of the Example

```
                //list(x)
                local e,f;
                (e,f) = open(C);
//list(x) * e|->(C{i},f) * f|->(C{i},e)
//(list(x)*e|->(C{i},f)) * (f|->(C{i},e))
```

```
local t;                       local y, e=0;
while (x != null) {            while (e == 0) {
  t = x->tl;                     { y = receive(cell, f);
  send(cell, e, x);                free(y);
  x = t;                           send(ack, f);
  receive(ack, e); }  ||         } + {
send(close_me, e, e);              e = receive(close_me, f);
// emp                           }}
                               close(e, f);
```

Proof of the Example

```
// f|->(C{i},e)
local x, e=0;

while (e == 0) {

  { x = receive(cell, f);

    dispose(x);

    send(ack, f);
  } + {
  e = receive(close_me, f);

  }
}

close(e, f);
```

Proof of the Example

```
// f|->(C{i},e)
local x, e=0;
// f|->(C{i},e) /| e=0
while (e == 0) {
  // f|->(C{i},e) /| e=0
  { x = receive(cell, f);
    // f|->(C{ack},e) * x |-> -
    dispose(x);
    // f|->(C{ack},e)
    send(ack, f);
  } + {
  e = receive(close_me, f);
  // f|->(C{end},e) * e|->(C{end},f)
  }
}
// f|->(C{end},e) * e|->(C{end},f)
close(e, f);
// emp
```

Proof of the Example

```
                 //list(x)
                 local e,f;
                 (e,f) = open(C);
//list(x) * e|->(C{i},f) * f|->(C{i},e)
//(list(x)*e|->(C{i},f)) * (f|->(C{i},e))
```

```
local t;                      local y, e=0;
while (x != null) {           while (e == 0) {
  t = x->tl;                    { y = receive(cell, f);
  send(cell, e, x);               free(y);
  x = t;                          send(ack, f);
  receive(ack, e); }   ||       } + {
send(close_me, e, e);             e = receive(close_me, f);
// emp                          }}
                              close(e, f);
```

Proof of the Example

```
                //list(x)
                local e,f;
                (e,f) = open(C);
//list(x) * e|->(C{i},f) * f|->(C{i},e)
//(list(x)*e|->(C{i},f)) * (f|->(C{i},e))
```

```
local t;                          local y, e=0;
while (x != null) {               while (e == 0) {
  t = x->tl;                        { y = receive(cell, f);
  send(cell, e, x);                   free(y);
  x = t;                              send(ack, f);
  receive(ack, e); }      ||        } + {
send(close_me, e, e);                 e = receive(close_me, f);
// emp                              }}
                                  close(e, f);
                                  // emp
```

```
                 //list(x)
                 local e,f;
                 (e,f) = open(C);
//list(x) * e|->(C{i},f) * f|->(C{i},e)
//(list(x)*e|->(C{i},f)) * (f|->(C{i},e))
```

```
local t;                        local y, e=0;
while (x != null) {             while (e == 0) {
  t = x->tl;                      { y = receive(cell, f);
  send(cell, e, x);                 free(y);
  x = t;                            send(ack, f);
  receive(ack, e); }      ||      } + {
send(close_me, e, e);               e = receive(close_me, f);
// emp                            }}
                                close(e, f);
                                // emp
```

```
                 // emp
```

In this Talk [APLAS'09]

- Formalization of heap-manipulating, message passing programs with contracts
- Contracts help us to ensure the absence of memory leaks
- Proof system
- Tool to prove specifications: Heap-Hop

In this Talk [APLAS'09]

- Formalization of heap-manipulating, message passing programs with contracts
- Contracts help us to ensure the absence of memory leaks
- Proof system
- Tool to prove specifications: Heap-Hop
- Not in this talk: semantics (based on abstract separation logic)

Conclusion

In this Talk [APLAS'09]

- Formalization of heap-manipulating, message passing programs with contracts
- Contracts help us to ensure the absence of memory leaks
- Proof system
- Tool to prove specifications: Heap-Hop
- Not in this talk: semantics (based on abstract separation logic)

In a Future Talk

- Contracts help us to ensure the absence of deadlocks
- Tackle real case studies: Singularity, MPI, distributed GC, ...