

# Proving Copyless Contract-Based Message Passing

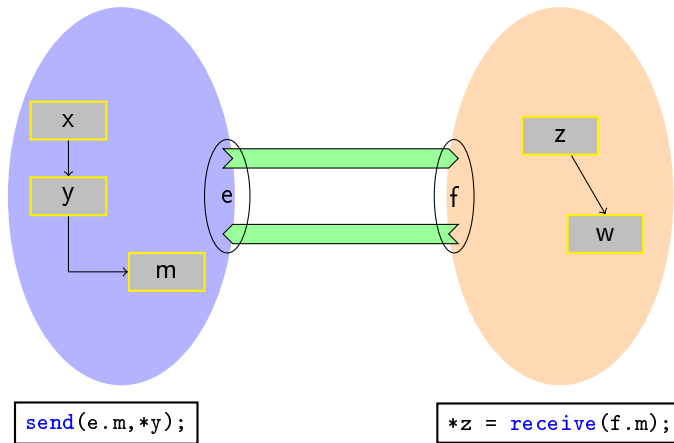
Jules Villard<sup>1</sup>   **Étienne Lozes**<sup>1</sup>   Cristiano Calcagno<sup>2</sup>

<sup>1</sup>LSV, ENS Cachan, CNRS

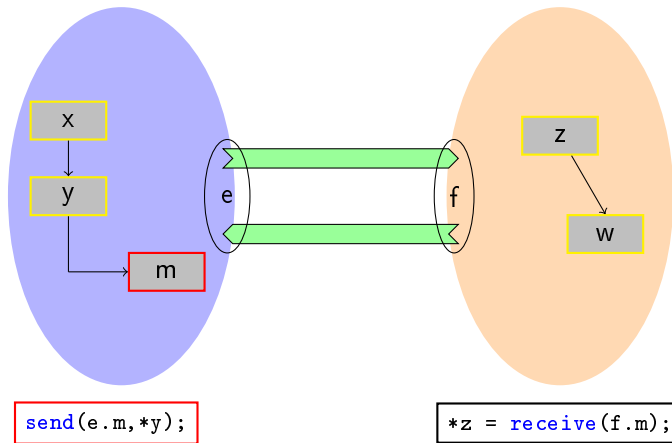
<sup>2</sup>Imperial College, London

Dagstuhl seminar – Typing, Analysis and Verification of  
Heap-Manipulating Programs

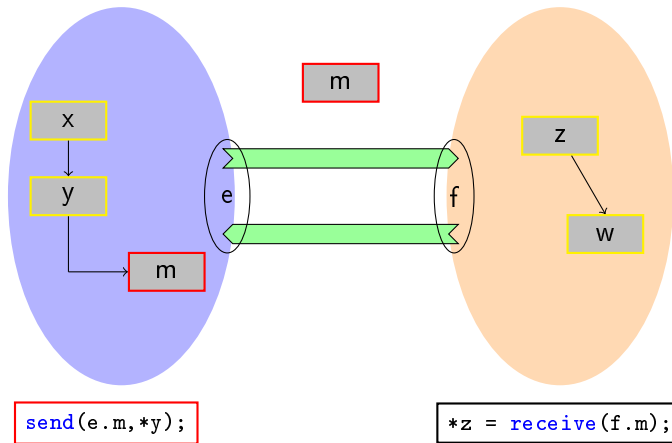
## Message Passing with copies (distributed memory)



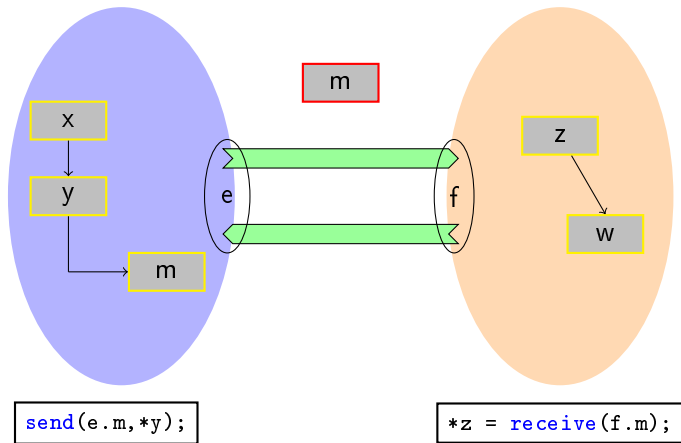
## Message Passing with copies (distributed memory)



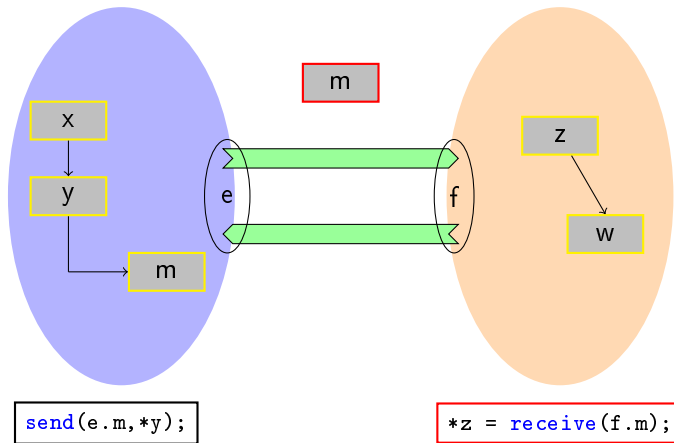
## Message Passing with copies (distributed memory)



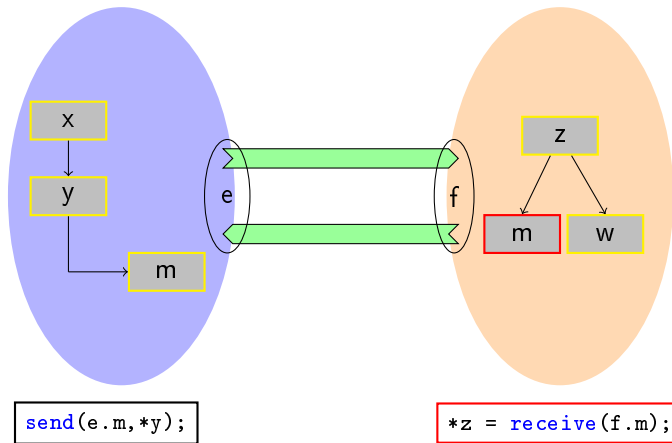
## Message Passing with copies (distributed memory)



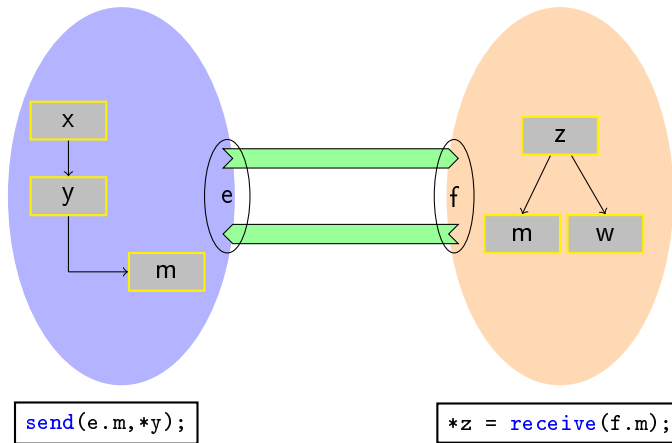
## Message Passing with copies (distributed memory)



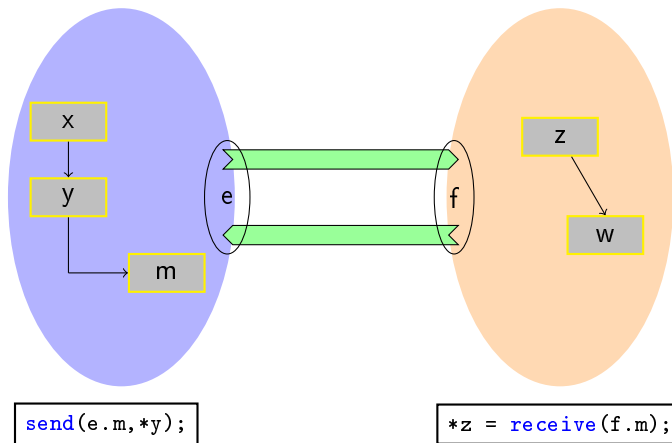
## Message Passing with copies (distributed memory)



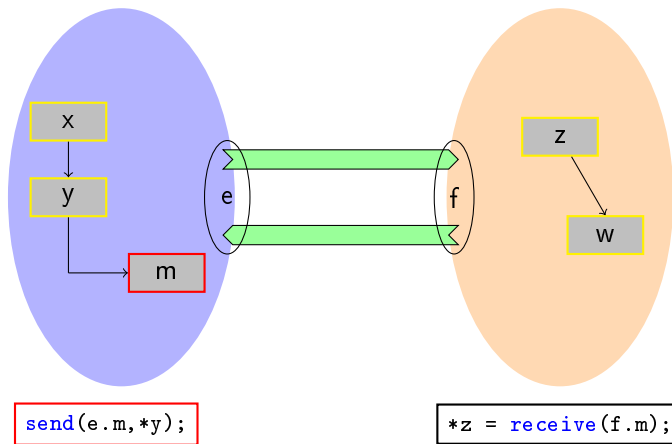
## Message Passing with copies (distributed memory)



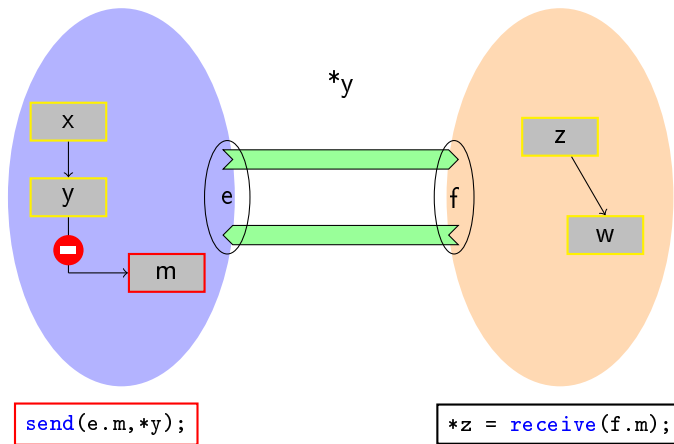
## Copyless Message Passing for a shared memory



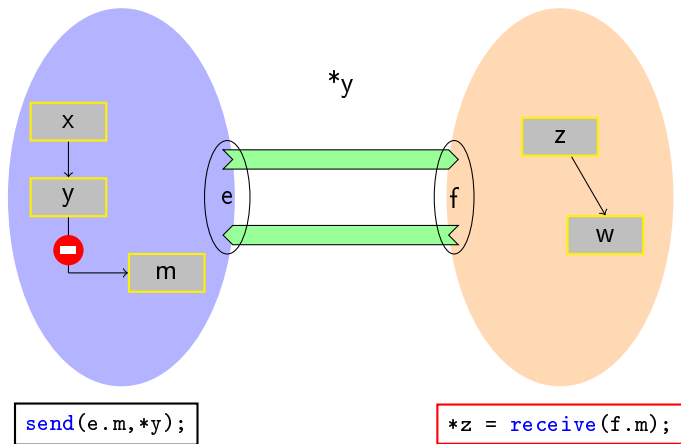
## Copyless Message Passing for a shared memory



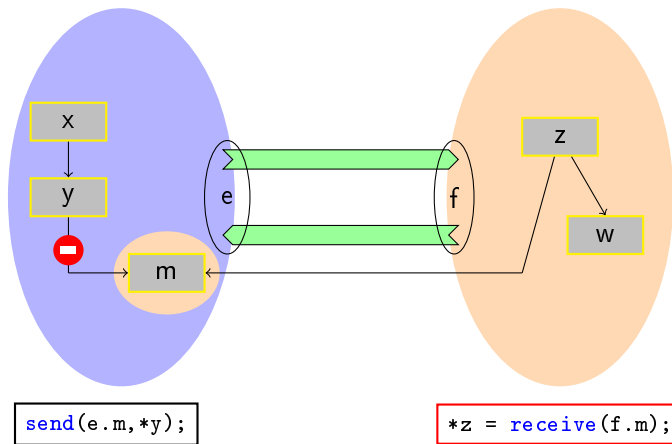
## Copyless Message Passing for a shared memory



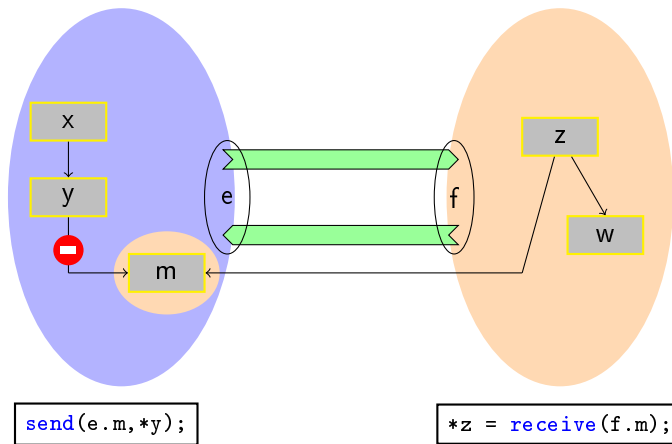
## Copyless Message Passing for a shared memory



## Copyless Message Passing for a shared memory



## Copyless Message Passing for a shared memory



## An implementation: Singularity

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
- ▶ Many guarantees ensured by the compiler:
  - **race freedom** (e.g. process isolation)
  - **progress**

## Sing# communication model

- ▶ Channels are **bidirectional** and **asynchronous**  
channel = pair of FIFO queues
- ▶ Channel is determined by a pair of **endpoints**  
similar to socket model
- ▶ Endpoints are allocated, disposed, and may be passed through channels  
under some conditions, similar to internal mobility in  $\pi$ -calculus
- ▶ Communications are ruled by user-defined **contracts**  
similar to session types

## Motivations

- ▶ Define a simple model of this language and eventually a (less simple) semantics.
- ▶ Provide a proof system based on Separation Logic
- ▶ Detail soundness:  
which properties are ensured by the proof system?

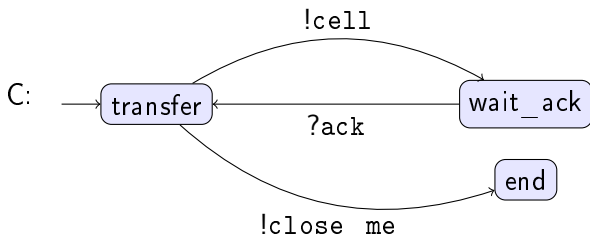
- ▶ Define a simple model of this language and eventually a (less simple) semantics.
- ▶ Provide a proof system based on Separation Logic
- ▶ Detail soundness:  
which properties are ensured by the proof system?
- ▶ Some questions:
  - how can one represent this form of ownership transfer in SL?

- ▶ Define a simple model of this language and eventually a (less simple) semantics.
- ▶ Provide a proof system based on Separation Logic
- ▶ Detail soundness:  
which properties are ensured by the proof system?
  
- ▶ Some questions:
  - how can one represent this form of ownership transfer in SL?
  - how Sing# static analysis mixes heap analysis and communications' analysis? what are the contracts good for?

- ▶ Define a simple model of this language and eventually a (less simple) semantics.
- ▶ Provide a proof system based on Separation Logic
- ▶ Detail soundness:  
which properties are ensured by the proof system?
  
- ▶ Some questions:
  - how can one represent this form of ownership transfer in SL?
  - how Sing# static analysis mixes heap analysis and communications' analysis? what are the contracts good for?
  - can we go beyond some limitations of Sing# (full mobility, garbage collection)?

## A Contract

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



# 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

$c ::= \text{assume}(B) \mid x := E$   
 $\quad \mid x := \text{new}() \mid \text{free}(x) \mid *E := E \mid x := *E \mid \dots$

## Programs

$p ::= c \mid p; p \mid p \parallel p \mid p + p \mid p^* \mid \text{local } x \text{ in } p$

# 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

$c ::= \text{assume}(B) \mid x := E$   
 $\quad \mid x := \text{new}() \mid \text{free}(x) \mid *E := E \mid x := *E \mid \dots$

## Programs

$p ::= c \mid p; p \mid p \parallel p \mid p + p \mid p^* \mid \text{local } x \text{ in } p$

Some simplifying choices:

- ▶ we will not distinguish internal/external choice (switch receive and if then else are modeled by +)
- ▶ we do not model fork/join constructs.

## Syntax of atomic commands (continued)

$c ::=$  ...

$(x,y) := \text{open}(C)$	(creates a channel with endpoints $x,y$ )
$\text{close}(E,E')$	(channel disposal)
$\text{send}(E.m, E')$	(sends message $m$ over endpoint $E$ )
$x := \text{receive}(E.m)$	(receives message $m$ over endpoint $E$ )

### Comments

- ▶  $m$  is a **message identifier**, not the value of the message
- ▶ usually, disposal is bipartite, involves an implicit message in asynchronous implementation.  
**closing message should be explicit in our setting.**

## A very simple example

```
local x,y in
  (x,y)=open(C);
  send(x.m,a);
  b:=receive(y.m);
  close(x,y);
```

$\approx$

```
b:=a;
```

## Sending a list cell by cell

Assume  $x$  is initially a pointer to some linked list.

```
local e,f;  
(e,f) := open(C);
```

```
local t;                                local y,e=null;  
while x != null {                       while e=null {  
  t := *x;                               switch receive{  
  send(e.cell, x);                       case y := receive(f.cell):  
  x := t;                                ||   free(y);  
  receive(e.ack); }                      send(f.ack);  
send(e.close_me, e);                    case e := receive(f.close_me): {}  
                                        }}  
                                        close(e, f);
```

## Proof system overview

$$\frac{\{A\} p \{B\}}{\{A * F\} p \{B * F\}}$$

$$\frac{\{A\} p \{B\} \quad \{A'\} p' \{B'\}}{\{A * A'\} p || p' \{B * B'\}}$$

+ other standard structural rules + **small axioms**

### We use abstract separation logic

- ▶ We may define a memory model  $\Sigma$  and a semantics of assertions  $\llbracket A \rrbracket \subseteq \Sigma$ .
- ▶ We may define a semantics  $\llbracket p \rrbracket : \Sigma \rightarrow \mathcal{P}(\Sigma)^\top$ , where  $\mathcal{P}(\Sigma)^\top \triangleq \mathcal{P}(\Sigma) \uplus \{\top\}$  and  $\top$  models fault (illegal dereferencing, contract violation,...).
- ▶  $\llbracket p \rrbracket$  is **local**: runtime race-checks, frame and parallel rule are sound for free.

# Soundness through local semantics

---

## Soundness' Theorem

If  $\{A\} p \{B\}$  is derivable then:

1.  $p$  is **safe** (does not fault by illegal dereferencing);
2.  $p$  is **race-free**;
3.  $p$  is **contract-obedient** (contract over-approximates control flow);
4.  $\llbracket p \rrbracket(\llbracket A \rrbracket) \subseteq \llbracket B \rrbracket$ .

## What the Soundness Theorem does not entail

- ▶ Proved programs terminate
- ▶ Proved programs are leak-free
- ▶ Proved programs are deadlock-free

## What the Soundness Theorem does not entail

- ▶ Proved programs terminate  
FALSE!!
- ▶ Proved programs are leak-free  
True, with some conditions
- ▶ Proved programs are deadlock-free  
True, with some conditions

## Why this form of soundness is not enough

### 1) The local semantics is not the intended one.

The local semantics modifies memory while transfers.

For instance, according to the local semantics:

```
(x, y) = open(C);  
send(x.m, a);  
b := receive(y.m);  
close(x, y);
```

$\neq$

```
b := a;
```

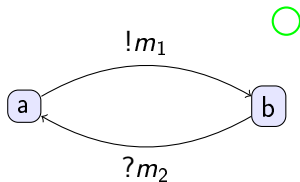
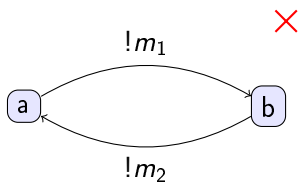
### 2) Contracts play an essential role for avoiding leaks and deadlocks.

# Properties of Contracts

Under some simple restrictions, contracts have interesting properties.

## Definition 1 (Synchronizing state)

Every cycle in the contract that goes through  $s$  must contain one send and one receive.



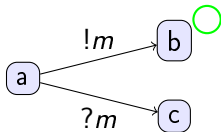
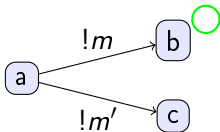
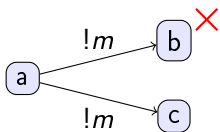
# Properties of Contracts

Under some simple restrictions, contracts have interesting properties.

## Definition 1 (Synchronizing state)

## Definition 2 (Determinism)

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



# Properties of Contracts

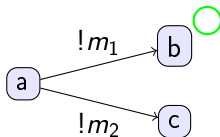
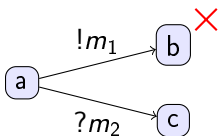
Under some simple restrictions, contracts have interesting properties.

Definition 1 (Synchronizing state)

Definition 2 (Determinism)

Definition 3 (Uniform choice)

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



# Properties of Contracts

Under some simple restrictions, contracts have interesting properties.

Definition 1 (Synchronizing state)

Definition 2 (Determinism)

Definition 3 (Uniform choice)

Lemma 4 (Half-Duplex)

2 & 3  $\Rightarrow$  *communications are half-duplex.*

Lemma 5 (Leak-free)

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

### Memory Leaks Theorem

If  $\{\text{emp}\} p \{\text{emp}\}$ , and if all contracts are leak-free, then  $p$  is leak-free.

### Progress Theorem

If  $\{A\} p \{B\}$ , all contracts are half-duplex, and all switch/receive are exhaustive, then  $p$  is deadlock-free.

Needs a new proof method: “Transfers erasure property”, e.g there is a global semantics that is sound, and that does not transfer memory.

## Comparison with Sing#

	<b>Sing#</b>	<b>Heap-hop</b>
Synchronizing states	all	final states
Mixed choice	yes?	no
Determinism	yes?	no
Mobility	internal mobility	full mobility
Communications	bounded channels	half-duplex
Disposal	GC	no memory leaks

## Standard SL formulae

$A ::=$	$\text{emp}_s \mid \text{own}(x) \mid E = E$	stack predicates
	$\mid \text{emp}_h \mid E \mapsto E$	heap predicates
	$\mid \neg A \mid A \wedge A \mid \exists X. A \mid A * A \mid A \multimap A$	operators

## Shorthands

- ▶  $O \Vdash A \triangleq (\text{own}(x_1) * \dots * \text{own}(x_n)) \wedge A$  for  $O = x_1, \dots, x_n$
- ▶  $E \mapsto - \triangleq \exists X. E \mapsto X$

## Standard deduction Rules

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

$$\frac{\{\text{own}(z) * A\} p[x \leftarrow z] \{\text{own}(z) * B\}}{\{A\} \text{local } x \text{ in } p \{B\}} \quad z \text{ fresh}$$

$$\{x \Vdash \text{emp}\} x := \text{new}() \{x \Vdash x \mapsto -\}$$

$$\{x, O \Vdash v \mapsto v' \wedge E = v\} x := *E \{x, O \Vdash v \mapsto v' \wedge x = v'\}$$

$$\{O \Vdash E \mapsto - \wedge F = v\} *E := F \{O \Vdash E \mapsto v\}$$

## Assertion Language (extension)

$A ::= \dots$   
 $| \text{emp}_{ep} \mid E \xrightarrow{\text{peer}}(C[a], E')$  endpoints' predicates

Intuitively  $E \xrightarrow{\text{peer}}(C[a], E')$  means :

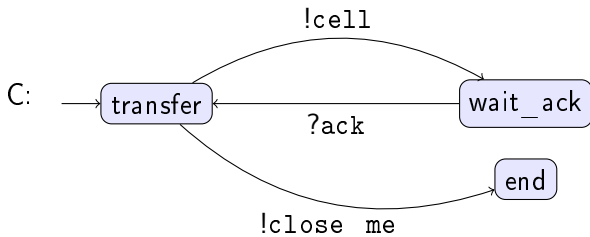
- ▶ E is an allocated endpoint,
- ▶ its peer is E',
- ▶ it is ruled by contract C,
- ▶ it currently is in contract's state  $a$ .

## Annotating 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

## Annotating Messages

```
contract C {  
  message ack      [emp];  
  message cell     [val|->X];  
  message close_me [val|-p->(C[end],X) /\ val=src];  
  
  initial state transfer {  
    !cell --> wait_ack;  
    !close_me --> end; };  
  state wait_ack { ?ack --> transfer; };  
  final state end {}; }
```



## Rules for Channel Manipulation

Open and Close rules:

$$\frac{i = \text{init}(C)}{\{e, f \Vdash \text{emp}\} (e, f) := \text{open}(C) \{e, f \Vdash e \xrightarrow{\text{peer}}(C[i], f) * f \xrightarrow{\text{peer}}(\bar{C}[i], e)\}}$$

$$\frac{f \in \text{final}(C)}{\{O \Vdash E \xrightarrow{\text{peer}}(C[f], E') * E' \xrightarrow{\text{peer}}(\bar{C}[f], E)\} \text{close}(E, E') \{O \Vdash \text{emp}\}}$$

# Rules for Channel Manipulation

Receive rule

$$\frac{a \xrightarrow{?m} b \in C}{\begin{array}{c} \{O, x \Vdash E \xrightarrow{peer}(C[a], X)\} \\ x := \text{receive}(E.m) \\ \{O, x \Vdash E \xrightarrow{peer}(C[b], X) * I_m(x, X)\} \end{array}}$$

## Rules for Channel Manipulation

Send rules

$$\frac{a \xrightarrow{!m} b \in C}{\{O \Vdash E \xrightarrow{\text{peer}} (C[a], -) * I_m(E', E)\} \text{ send}(E.m, E') \{O \Vdash E \xrightarrow{\text{peer}} (C[b], -)\}}$$

$$\frac{a \xrightarrow{!m} b \in C}{\{O \Vdash E \xrightarrow{\text{peer}} (C[a], -) * (E \xrightarrow{\text{peer}} (C[b], -) \multimap I_m(E', E))\} \text{ send}(E.m, E') \{O \Vdash \text{emp}\}}$$

## Proof of the Example

```
// x ||- ls(x)
local e,f;
// x,e,f ||- ls(x)
(e,f) := open(C);
//x,e,f||-ls(x) * e|-p->(C[i],f) * f|-p->(C[i],e)
//(x,e||-ls(x)*e|-p->(C[i],-)) * (f||-f|-p->(C[i],-))
```

---

```
local t;                local y,e=null;
while x != null {      while e=null {
  t := *x;              { y := receive(f.cell);
  send(e.cell, x);      free(y);
  x := t;                send(f.ack);
  receive(e.ack); } || } + {
send(e.close_me, e);    e := receive(f.close_me);
                        }}
                        close(e, f);
```

---

## Proof of the Example

```
// x,e ||- ls(x) * e|-p->(C[i],-)
local t;

while x != null {

    t := *x;

    send(e.cell, x);

    x := t;
    receive(e.ack); }

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

## Proof of the Example

---

```
// x,e ||- ls(x) * e|-p->(C[i],-)
local t;
// x,e,t ||- ls(x) * e|-p->(C[i],-)
while x != null {
  // 0||- x|-> Y * ls(Y) * e|-p->(C[i],-)
  t := *x;
  // idem || t=Y
  send(e.cell, x);
  // 0||- ls(t) * e|-p->(C[ack],-)
  x := t;
  receive(e.ack); }
// 0||- e|-p->(C[transfer],-)
send(e.close_me, e);
// 0||- emp
```

## Proof of the Example

```
// x ||- ls(x)
local e, f;
// x, e, f ||- ls(x)
(e, f) := open(C);
//(x, e ||- ls(x) * e | - p -> (C[i], -)) * (f ||- f | - p -> (C[i], -))
```

---

```
local t;                local y, e=null;
while x != null {      while e=null {
  t := *x;              { y := receive(f.cell);
  send(e.cell, x);      free(y);
  x := t;               send(f.ack);
  receive(e.ack); } || } + {
send(e.close_me, e);   e := receive(f.close_me);
                       }}
                       close(e, f);
```

---

## Proof of the Example

```
// x ||- ls(x)
local e,f;
// x,e,f ||- ls(x)
(e,f) := open(C);
//(x,e||-ls(x)*e|-p->(C[i],-)) * (f||-f|-p->(C[i],-))

```

---

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

```

---

## Proof of the Example

---

```
// f||- f|-p->(C[i],-)
local x,e=null;

while e=null {

  { x := receive(f.cell);

    free(x);

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

}
}

close(e, f);
```

## Proof of the Example

---

```
// f||- f|-p->(C[i],-)
local x,e=null;
// f,x,e ||- f|-p->(C[i],-) /| e=0
while e=null {
  // 0||- f|-p->(C[i],-) /| e=0
  { x := receive(f.cell);
    // 0||- f|-p->(C[ack],-) * x |-> -
    free(x);
    // 0||- f|-p->(C[ack],-)
    send(f.ack);
  } + {
    e := receive(f.close_me);
    // 0||- f|-p->(C[end],e) * e|-p->(C[end],f)
  }
}
// 0||- f|-p->(C[end],e) * e|-p->(C[end],f)
close(e, f);
// 0||- emp
```

## Proof of the Example

```
// x ||- ls(x)
local e,f;
// x,e,f ||- ls(x)
(e,f) := open(C);
//(x,e||-ls(x)*e|-p->(C[i],-)) * (f||-f|-p->(C[i],-))

```

---

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

```

---

## Proof of the Example

```
// x ||- ls(x)
local e,f;
// x,e,f ||- ls(x)
(e,f) := open(C);
//(x,e||-ls(x)*e|-p->(C[i],-)) * (f||-f|-p->(C[i],-))

```

---

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

```

---

## Proof of the Example

```
// x ||- ls(x)
local e,f;
// x,e,f ||- ls(x)
(e,f) := open(C);
//(x,e||-ls(x)*e|-p->(C[i],-)) * (f||-f|-p->(C[i],-))

```

---

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

```

---

```
// x ||- emp
```

## Conclusion

- ▶ We have formalized a language featuring copyless message passing with contracts and explicit memory manipulation.
- ▶ Our proof system allows us to verify that programs are safe, race-free, deadlock-free and leak-free in modular way.
- ▶ We proved soundness using abstract separation logic.
- ▶ The leak-free property requires some extra work on a **non-transferring semantics**.
- ▶ We have a tool and some small case studies.

## Future Work:

- ▶ Develop heaphop and examples.
- ▶ Real case studies: Singularity, MPI, distributed GC,... log files?
- ▶ Extend contract's theory.
- ▶ Load-balancing challenge