Heaps and Hops
Soutenance de thèse

Jules Villard

LSV, ENS Cachan, CNRS
Moore’s Law

*The number of transistors one can put on a chip doubles every two years*
## Shift towards Concurrency

### Moore’s Law

*The number of transistors one can put on a chip doubles every two years*

### Moore’s law until recently

The frequency of processors doubles every two years
### Shift towards Concurrency

#### Moore’s Law

*The number of transistors one can put on a chip doubles every two years*

#### Moore’s law until recently

The frequency of processors doubles every two years

#### Moore’s law nowadays

- The frequency of processors is reaching limits
- Augment the number of processors on a chip!
## Shift towards Concurrency

### Moore’s Law

*The number of transistors one can put on a chip doubles every two years*

### Moore’s law until recently

The frequency of processors doubles every two years

### Moore’s law nowadays

- The frequency of processors is reaching limits
- Augment the number of processors on a chip!

- Concurrent programs are more needed than ever
- They are hard to write **correctly** and **efficiently**
Message Passing in Multicore Systems

- New paradigm: message passing over a shared memory
- Leads to efficient, copyless message passing
- May be more error-prone
### To Copy or not to Copy?

**Copyful**

<table>
<thead>
<tr>
<th>data</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(struct,e,data);</td>
<td>d = receive(struct,f);</td>
</tr>
</tbody>
</table>

- (e, f): channel
- data points to a big struct
- struct: type of message

---

**Introduction**

- **Concurrency**

---

---
To Copy or not to Copy?

Copyful

<table>
<thead>
<tr>
<th>data</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(struct,e,data);</td>
<td>d = receive(struct,f);</td>
</tr>
</tbody>
</table>

- (e, f): channel
- data points to a big struct
- struct: type of message
### To Copy or not to Copy?

#### Copyful

<table>
<thead>
<tr>
<th>data</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{send}($struct,e,data$)$</td>
<td>$d = \text{receive}($struct,f$)$</td>
</tr>
</tbody>
</table>

#### Copyless

<table>
<thead>
<tr>
<th>data</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{send}($pointer,e,data$)$</td>
<td>$d = \text{receive}($pointer,f$)$</td>
</tr>
</tbody>
</table>
### To Copy or not to Copy?

#### Copyful

<table>
<thead>
<tr>
<th>Data (struct, e, data)</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send(struct, e, data);</code></td>
<td><code>d = receive(struct, f);</code></td>
</tr>
</tbody>
</table>

#### Copyless

<table>
<thead>
<tr>
<th>Data (pointer, e, data)</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send(pointer, e, data);</code></td>
<td><code>d = receive(pointer, f);</code></td>
</tr>
</tbody>
</table>
To Copy or not to Copy?

Copyful

data → send(struct, e, data); → d = receive(struct, f); →

Copyless

data → send(pointer, e, data); → dispose(data); →
d = receive(pointer, f); → dispose(d); →

Race!
To Copy or not to Copy?

Copyful

```
data     d
send(struct,e,data);   d = receive(struct,f);
```

Copyless

```
data     d
send(pointer,e,data);  d = receive(pointer,f);
dispose(data);  dispose(d);
```
To Copy or not to Copy?

Copyful

data → 

send(struct, e, data);

Copyless No race

data

d →

d = receive(struct, f);

dispose(d);

Introduction • Concurrency
Singularity: a research project and an operating system.

- No hardware memory protection
- Sing\# language
- Isolation is verified at compile time
- Invariant: each memory cell is owned by at most one thread
- No shared resources
- Copyless message passing
Singularity: a research project and an operating system.

- No hardware memory protection
- Sing\# language
- Isolation is verified at compile time
- Invariant: each memory cell is owned by at most one thread
- No shared resources
- Copyless message passing
Singularity Features

[Fähndrich et al. ’06]

- Channels are **bidirectional** and **asynchronous**
  channel = pair of FIFO queues
- Channels are made of two **endpoints**
  similar to the socket model
- Endpoints can be allocated, disposed of, and communicated through channels
  similar to the $\pi$-calculus
- Communications are ruled by user-defined **contracts**
  similar to session types
- **No formalisation**
  How to ensure the absence of bugs?

Introduction • Concurrency
Formal Verification

- **Model** of the program
- **Specify** a correctness criterion in a mathematical language
- **Prove** a theorem which links the two
Main Contributions of the Thesis

- **Model** of the program
  - Semantics of copyless message passing programs

- **Specify** a correctness criterion in a mathematical language
  - Hoare triples: separation logic for channels in the heap
  - Contracts

- **Prove** a theorem which links the two
  - Automatic tool: Heap-Hop
  - Extend the proof system of separation logic
  - Properties of contracts rub off on programs
Our Analysis

Model

Specify

SL+MP

Program

Heap-Hop

Proof

Contracts Prop.

= Program Prop.

Contracts

Program

= Proof

Contracts Prop.

/\
Heap-Hop

Program

- message passing primitives

Proof

+ 

Contracts Prop.

= 

Program Prop.

SL+MP

Contracts

Heap-Hop
Message Passing Primitives

- \((e, f) = \text{open}()\) Creates a bidirectional channel between endpoints \(e\) and \(f\)
- \(\text{close}(e, f)\) Closes the channel \((e, f)\)
- \(\text{send}(a, e, x)\) Sends message starting with value \(x\) on endpoint \(e\). The message has type/tag \(a\)
- \(x = \text{receive}(a, e)\) Receives message of type \(a\) on endpoint \(e\) and stores its value in \(x\)

```haskell
set_to_ten(x) {
    local e,f;
    (e,f) = open();
    send(integer,e,10);
    x = receive(integer,f);
    close(e,f);
}
```
Switch Receive

- **switch receive** selects a receive branch depending on availability of messages

```plaintext
if ( x ) {
    send (cell, e, x);
} else {
    send (integer, e, 0);
}

switch receive {
    y = receive (cell, f): { dispose (y); }
    z = receive (integer, f): {}
}
```
Program $\rightarrow$ Proof $\rightarrow$ SL+MP

Contracts $\rightarrow$ Proof $\rightarrow$ Contracts Prop.

Program Prop.

- Race freedom
- Reception fault freedom
- Leak freedom
Safety Properties

Separation property

At each point in the execution, the state can be partitioned into what is owned by each program and each message in transit.

- Programs access only what they own.
- Prevents races.
Safety Properties

Separation property

At each point in the execution, the state can be \textit{partitioned} into what is owned by each program and each message in transit.

- Programs access only what they own.
- Prevents races.
Safety Properties

Separation property

At each point in the execution, the state can be **partitioned** into what is owned by each program and each message in transit.

- Programs access only what they own.
- Prevents races.
Safety Properties

Separation property

At each point in the execution, the state can be partitioned into what is owned by each program and each message in transit.

- Programs access only what they own.
- Prevents races.
Safety Properties

Separation property

Invalid receptions freedom

```
switch receive { 
    y = receive(a,f): { ... }
    z = receive(b,f): { ... }
}
```

```
send(c,e,x);
```

...
Safety Properties

Separation property

Invalid receptions freedom

Leak freedom

The program does not leak memory.

```c
1 main() {
2   local x,e,f;
3
4   x = new();
5   (e,f) = open();
6   send(cell,e,x);
7   close(e,f);
8 }
```
Heap-Hop

Program → Proof + Contracts Prop. = Program Prop.

SL+MP

Contracts

- Communicating automata
A Dialogue System

- Sending transitions: !a
- Receiving transitions: ?a
- Two buffers: one in each direction
- Configuration: \( \langle q, q', w, w' \rangle \)
A Dialogue System

Channel Contracts • Communicating Automata

\[ \langle q, q_0, \varepsilon, \varepsilon \rangle \]
A Dialogue System

Channel Contracts • Communicating Automata
A Dialogue System

Channel Contracts • Communicating Automata

\[ \langle q, q_2, ab, \varepsilon \rangle \]
A Dialogue System

\[ \langle q_a, q_2, b, \varepsilon \rangle \]
A Dialogue System

\[ \langle q, q_2, b, a \rangle \]
A Dialogue System

\[
\langle q, q_3, b, \varepsilon \rangle
\]
A Dialogue System

Channel Contracts • Communicating Automata
A Dialogue System

\[ \langle q, q_3, \varepsilon, b \rangle \]
A Dialogue System

Channel Contracts • Communicating Automata
Describe dual communicating finite state machines

\[ e \xrightarrow{\text{init}} \xrightarrow{!\text{pointer}} \xrightarrow{\text{end}} \]
Describe dual communicating finite state machines

\[
\begin{array}{c}
\mathcal{C} \\
\mathcal{C}'
\end{array}
\]

\[
\begin{array}{c}
\text{init} \quad \text{!pointer} \quad \text{end} \\
\text{init} \quad \text{?pointer} \quad \text{end}
\end{array}
\]
Describe dual communicating finite state machines

\[ \mathcal{C} \quad \text{init} \xrightarrow{!\text{pointer}} \text{end} \quad \mathcal{C}' \]

\[ \mathcal{C}' \quad q' \xrightarrow{!\text{cell}} q \xrightarrow{!\text{fin}} \text{end} \quad \mathcal{C}' \quad q' \xleftarrow{?\text{cell}} q \xrightarrow{?\text{fin}} \text{end} \]
Contracts as Protocol Specifications

- \((e,f) = \text{open}(C)\): initialise endpoints in the initial state of the contract
- \(\text{send}(a,e,x)\): becomes a !\(a\) transition
- \(y = \text{receive}(a,f)\): becomes a ?\(a\) transition
- \(\text{closed}(e,f)\) only when both endpoints are in the same **final** state.
Heap-Hop

Program → Proof + SL+MP

Contracts Prop. = Program Prop.

- Reception faults
- Leaks
Reception Errors

Definition

\[ \langle q_1, q_2, a \cdot w_1, w_2 \rangle \text{ is a reception fault if} \]

- \( q_1 \xrightarrow{?b} q \) for some \( b \) and \( q \) and
- \( \forall b, q. q_1 \xrightarrow{?b} q \) implies \( b \neq a \)

\[ \langle q, q, \varepsilon, \varepsilon \rangle \]
Reception Errors

Definition

$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ is a **reception fault** if

- $q_1 \xrightarrow{?b} q$ for some $b$ and $q$ and
- $\forall b, q. \quad q_1 \xrightarrow{?b} q$ implies $b \neq a$

\[ \langle q_1, q, a, \varepsilon \rangle \]
Reception Errors

Definition

\( \langle q_1, q_2, a \cdot w_1, w_2 \rangle \) is a **reception fault** if

- \( q_1 \xrightarrow{?b} q \) for some \( b \) and \( q \) and
- \( \forall b, q, q_1 \xrightarrow{?b} q \implies b \neq a \)

\[ \langle q_1, q'_1, a, b \rangle \]
Reception Errors

Definition

\[ \langle q_1, q_2, a \cdot w_1, w_2 \rangle \text{ is a reception fault if} \]

- \( q_1 \xrightarrow{?b} q \) for some \( b \) and \( q \) and
- \( \forall b, q. \, q_1 \xrightarrow{?b} q \) implies \( b \neq a \)

\[ \langle q_1, q_1', a, b \rangle \xrightarrow{?b} \text{ error} \]
Reception Errors

Definition

\[
\langle q_1, q_2, a \cdot w_1, w_2 \rangle \text{ is a reception fault if }
\]

- \( q_1 \xrightarrow{?b} q \) for some \( b \) and \( q \) and
- \( \forall b, q. \ q_1 \xrightarrow{?b} q \implies b \neq a \)

- A contract is reception fault-free if it cannot reach a reception fault.
Undelivered Messages

Definition

\[ \langle q_f, q_f, w_1, w_2 \rangle \text{ is a leak} \text{ if } w_1 \cdot w_2 \neq \varepsilon \text{ and } q_f \text{ is final.} \]

\[ \langle q, q, \varepsilon, \varepsilon \rangle \]
Undelivered Messages

Definition

\[ \langle q_f, q_f, w_1, w_2 \rangle \text{ is a leak if } w_1 \cdot w_2 \neq \varepsilon \text{ and } q_f \text{ is final.} \]

\[ \langle q_1, q, a, \varepsilon \rangle \]

Leak
### Definition

\[ \langle q_f, q_f, w_1, w_2 \rangle \text{ is a leak if } w_1 \cdot w_2 \neq \varepsilon \text{ and } q_f \text{ is final.} \]

### Leak

\[ \langle q_2, q, aa, \varepsilon \rangle \]
Undelivered Messages

Definition

\[ \langle q_f, q_f, w_1, w_2 \rangle \] is a **leak** if \( w_1 \cdot w_2 \neq \varepsilon \) and \( q_f \) is final.

Leak

\[ \langle q_2, q_2, a, \varepsilon \rangle \]
Definition

\( \langle q_f, q_f, w_1, w_2 \rangle \) is a \textbf{leak} if \( w_1 \cdot w_2 \neq \varepsilon \) and \( q_f \) is final.

\[
\langle q_2, q_2, a, \varepsilon \rangle
\]
Undelivered Messages

<table>
<thead>
<tr>
<th>Definition</th>
<th>Leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle q_f, q_f, w_1, w_2 \rangle$ is a <strong>leak</strong> if $w_1 \cdot w_2 \neq \varepsilon$ and $q_f$ is final.</td>
<td></td>
</tr>
</tbody>
</table>

- A contract is **leak free** if it cannot reach a leak.
- A contract is **safe** if it is reception fault free and leak free.
Safety of communicating systems is undecidable in general

Channel’s buffer \(\approx\) Turing machine’s tape
Safety of communicating systems is undecidable in general

Channel’s buffer ≈ Turing machine’s tape

Contracts are restricted (dual systems)
Safety of communicating systems is undecidable in general

Channel’s buffer $\approx$ Turing machine’s tape

- Contracts are restricted (dual systems)
- Contracts can encode Turing machines as well

**Theorem**

Safety is undecidable for contracts.
Safety of communicating systems is undecidable in general

Channel’s buffer \(\approx\) Turing machine’s tape

- Contracts are restricted (dual systems)
- Contracts can encode Turing machines as well

Theorem

Safety is undecidable for contracts.

- We give sufficient conditions for safety.
Sufficient Conditions for Reception Safety

**Definition**

**Deterministic contract**

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

- ![Diagram 1](image1)
- ![Diagram 2](image2)
- ![Diagram 3](image3)
## Sufficient Conditions for Reception Safety

<table>
<thead>
<tr>
<th>Definition</th>
<th>Deterministic contract</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition</strong></td>
<td>Positional contracts</td>
</tr>
<tr>
<td>All outgoing edges from a same state in a contract must be either all sends or all receives.</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
q \\
\downarrow \quad \downarrow \\
q_1 & q_2 \\
!a_1 & ?a_2
\end{array}
\quad \text{X}
\quad \begin{array}{c}
q \\
\downarrow \quad \downarrow \\
q_1 & q_2 \\
!a_1 & !a_2
\end{array}
\quad \text{○}
\]

Channel Contracts • Singularity Contracts
Sufficient Conditions for Reception Safety

**Definition**

**Deterministic contract**

**Definition**

**Positional contracts**

**Theorem**

[Stengel & Bultan’09] • [V., Lozes & Calcagno ’09]

Deterministic positional contracts are **reception fault free**.

Channel Contracts • Singularity Contracts
Sufficient Conditions for Reception Safety

Definition

Deterministic contract

Definition

Positional contracts

Theorem

[Stengel & Bultan’09] • [V., Lozes & Calcagno ’09]

Deterministic positional contracts are reception fault free.
Sufficient Conditions for Reception Safety

**Definition**
- Deterministic contract

**Definition**
- Positional contracts

**Theorem**

[Stengel & Bultan’09] • [V., Lozes & Calcagno ’09]

**Deterministic positional contracts are reception fault free.**

Channel Contracts • Singularity Contracts
Sufficient Conditions for Reception Safety

**Definition**
- Deterministic contract
- Positional contracts

**Theorem**
[Stengel & Bultan’09] • [V., Lozes & Calcagno ’09]

*Deterministic positional contracts are reception fault free.*
### Sufficient Conditions for Reception Safety

<table>
<thead>
<tr>
<th>Definition</th>
<th>Deterministic contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>Positional contracts</td>
</tr>
<tr>
<td>Theorem</td>
<td>[Stengel &amp; Bultan’09] ● [V., Lozes &amp; Calcagno ’09]</td>
</tr>
<tr>
<td><strong>Deterministic positional contracts are reception fault free.</strong></td>
<td></td>
</tr>
</tbody>
</table>

- **Channel Contracts**
  - **Singularity Contracts**

- **Diagram:**
  - State transitions and conditions for reception safety.
Another Source of Leaks

Channel Contracts • Singularity Contracts
Another Source of Leaks

\[ \langle q, q, \varepsilon, \varepsilon \rangle \]
Another Source of Leaks

\[ \langle q, q, a, \varepsilon \rangle \]
Another Source of Leaks

\[ \langle q, q, aa, \varepsilon \rangle \]
Another Source of Leaks

\[ \langle q, q, aaa, \varepsilon \rangle \]
**Synchronising Contracts**

<table>
<thead>
<tr>
<th>Definition</th>
<th>Synchronising state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A state $s$ is synchronising if every cycle that goes through it contains at least one send and one receive.</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image.png)

**Definition**

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

**Theorem** [V., Lozes & Calcagno '09]

Deterministic, positional and synchronising contracts are safe (fault and leak free).

**Channel Contracts**

- Singularity Contracts

---

* Channel Contracts  •  Singularity Contracts  

---
### Synchronising Contracts

<table>
<thead>
<tr>
<th>Definition</th>
<th>Synchronising state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A state $s$ is synchronising if every cycle that goes through it contains at least one send and one receive.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
<th>Synchronising contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>A contract is synchronising if all its final states are.</td>
<td></td>
</tr>
</tbody>
</table>
### Synchronising Contracts

<table>
<thead>
<tr>
<th>Definition</th>
<th>Synchronising state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A state $s$ is synchronising if every cycle that goes through it contains at least one send and one receive.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
<th>Synchronising contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>A contract is synchronising if all its final states are.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Theorem</th>
<th>[V., Lozes &amp; Calcagno ’09]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Deterministic, positional and synchronising contracts are safe (fault and leak free).</strong></td>
<td></td>
</tr>
</tbody>
</table>
Singularity Contracts

**Definition**

Singularity contracts are deterministic and **all** their states are synchronising.

- This is missing the positional condition!
- Does not guarantee reception fault freedom
- In fact, we proved that safety is still **undecidable** for deterministic or positional contracts.
- Positional Singularity contracts are **safe** and **bounded**.
Heap-Hop

Program

Proof

Contracts Prop.

Program Prop.

Contracts

SL+MP

• Extension to message passing
## Separation Logic

[Reynolds 02, O’Hearn 01, …]

<table>
<thead>
<tr>
<th>Separation Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>An <strong>assertion language</strong> to describe states</td>
</tr>
<tr>
<td>A <strong>proof system</strong> for Hoare triples</td>
</tr>
</tbody>
</table>

- Local reasoning for heap-manipulating programs
- Naturally describes ownership transfers
- Has been extended to storable locks [Gotsman et al. 07]
## Assertions

### Syntax

| $E$  | $\ ::= $ | $x \mid n \in \{0, 1, 2, \ldots \} \mid \cdots$ | expressions       |
| $\phi$ | $\ ::= $ | $E_1 = E_2 \mid E_1 \neq E_2 \mid \text{emp} \mid E_1 \leftrightarrow E_2 \mid \exists x. \phi \mid \phi_1 \land \phi_2 \mid \neg \phi \mid \phi_1 \ast \phi_2$ | stack predicates  |
|       |            |                     | heap predicates   |
|       |            |                     | formulas          |
Syntax (continued)

<table>
<thead>
<tr>
<th>( \phi ) ::= \ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \leftrightarrow (\mathcal{C}{q}, E') ) endpoint predicate</td>
</tr>
</tbody>
</table>

Intuitively \( E \leftrightarrow (\mathcal{C}\{q\}, E') \) means:

- \( E \) is an allocated endpoint
- it is ruled by contract \( \mathcal{C} \)
- it is currently in the control state \( q \) of \( \mathcal{C} \)
- its peer is \( E' \)
Heap-Hop

Program \rightarrow \text{Proof} \rightarrow \text{SL+MP}

\text{Contracts Prop.} \rightarrow \text{Program Prop.}

- Extends Smallfoot with message passing
- Written in OCaml
- Open source
[V., Lozes & Calcagno TACAS’10]
Heap-Hop

Program

Proof

Contracts

SL+MP

Contracts Prop.

Program Prop.

- rules for message passing
- message footprints

Heap-Hop
## Memory States $\sigma$

A memory state $\sigma$ has three components

- A variable valuation (stack)
- A heap for memory cells
- Buffers for endpoints

## Semantics of programs

Small-step interleaving operational semantics for programs $p$:

- $p, \sigma \rightarrow^* p', \sigma'$ (intermediate state)
- $p, \sigma \rightarrow^* \sigma'$ (final state)
- $p, \sigma \rightarrow^* \text{error}$ (error state)
\{ \phi \} \ p \ \{ \psi \} : \text{Hoare triple}

- \phi, \psi : \text{formulas}
- p : \text{program}

**Fault-free** interpretation of Hoare triples

If \{ \phi \} \ p \ \{ \psi \} is provable, then for all state \( \sigma \models \phi \),

1. \( p \) has no race or memory faults from \( \sigma \)
2. \( p \) implements its contracts
3. if \( p, \sigma \rightarrow^* \sigma' \) then \( \sigma' \models \psi \)

**Proof system**

Derivation rules to **prove** Hoare triples.
Communication Rules

**OPEN**

\[
i = \text{init}(C) \\quad \{\text{emp}\} \ (e,f) = \text{open}(C) \ \{e \mapsto (C\{i\},f) \ast f \mapsto (C\{i\},e)\}
\]

**CLOSE**

\[
q \in \text{finals}(C) \quad \{e \mapsto (C\{q\},f) \ast f \mapsto (\neg C\{q\},e)\} \ \text{close}(e,f) \ \{\text{emp}\}
\]

**SEND**

\[
q \xrightarrow{!a} q' \in C \quad e \mapsto (C\{q'\},-) \ast \phi \Rightarrow \gamma_a(e,x) \ast \phi' \\quad \{e \mapsto (C\{q\},-) \ast \phi\} \ \text{send}(a,e,x) \ \{\phi'\}
\]

**RECEIVE**

\[
q \xrightarrow{?a} q' \in C \quad \{e \mapsto (C\{q\},X')\} \ x = \text{receive}(a,e) \ \{e \mapsto (C\{q'\},X') \ast \gamma_a(X',x)\}
\]
**Communication Rules**

**Open**

\[ t = \text{init}(e) \]

\{emp\} (e,f) = open(C) (e = (C[q], f) \rightarrow f = (C[q], e))

**CLOSE**

\[ \text{q} \in \text{finals}(C) \]

\{e → (C[q], f) \# f → (¬C[q], e)\} close(e,f) \{emp\}

**SEND**

\[ q \rightarrow q' \in C \quad e = (C[q'], e) \rightarrow d = \text{send}(a,e,x) \rightarrow \]

\{e → (C[q'], X')\} x = receive(a,e) \{e → (C[q'], X') \# γ(a)(X', x)\}

**RECEIVE**

\[ q \xrightarrow{a} q' \in C \]

\{e → (C[q], X')\} x = receive(a,e) \{e → (C[q'], X') \# γ(a)(X', x)\}
Closing a Channel

\[
\text{CLOSE}
\]

\[
q \in \text{finals}(\mathcal{C})
\]

\[
\{ e \mapsto (\mathcal{C}\{ q \}, f) \} \star f \mapsto (\sim \mathcal{C}\{ q \}, e) \} \text{ close}(e, f) \{ \text{emp} \}
\]
Closing a Channel

\[
\text{CLOSE} \quad q \in \text{finals}(\mathcal{C}) \\
\{ e \mapsto (\mathcal{C}\{q\}, f) \ast f \mapsto (\sim \mathcal{C}\{q\}, e) \} \quad \text{close}(e, f) \{ \text{emp} \}
\]

Proving Copyless Message Passing • Proof System
**General Rule for Receive**

\[ \text{Receive} \]

\[ q \overset{?a}{\rightarrow} q' \in \mathcal{C} \]

\[ \{ e \mapsto (\mathcal{C}\{q\}, X') \} x = \text{receive}(a, e) \{ e \mapsto (\mathcal{C}\{q'\}, X') \ast \gamma_a(X', x) \} \]
General Rule for Receive

\[
\begin{align*}
q \xrightarrow{?a} q' \in C \\
\{ e \mapsto (C\{q\}, X') \} \ x = \text{receive}(a, e) \ \{ e \mapsto (C\{q'\}, X') \star \gamma_a(X', x) \}
\end{align*}
\]
General Rule for Receive

\[ \text{RECEIVE} \]

\[ q \overset{a}{\rightarrow} q' \in \mathcal{C} \]

\[ \{ e \mapsto (\mathcal{C}\{q\}, X') \} x = \text{receive}(a, e) \{ e \mapsto (\mathcal{C}\{q'\}, X') \ast \gamma_a(X', x) \} \]

Can be instantiated for each example:

\[ \gamma_{\text{cell}}(\text{src}, \text{val}) \triangleq \text{val} \mapsto - \]
\[ \gamma_{\text{ep}}(\text{src}, \text{val}) \triangleq \text{val} \mapsto (\mathcal{C}\{\text{end}\}, -) \land \text{val} = \text{src} \]
Heap-Hop

Program → Proof

• soundness

SL+MP → Proof

+

Contracts Prop. = Program Prop.

Contracts
### Definition

#### Program validity

\(\{\phi\} p \{\psi\}\) is valid if, for all \(\sigma \models \phi\):

- \(p\) has **no race or memory fault** starting from \(\sigma\)
- \(p\) has **no reception faults** starting from \(\sigma\)
- if \(p, \sigma \rightarrow^{*} \sigma'\) then \(\sigma' \models \psi\)

### Definition

#### Leak free programs

\(p\) is **leak free** if for all \(\sigma\)

\(p, \sigma \rightarrow^{*} \sigma'\) implies that the heap and buffers of \(\sigma'\) are empty
### Properties of Proved Programs

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Soundness</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ( { \phi } p { \psi } ) is provable with <em>reception fault free</em> contracts then ( { \phi } p { \psi } ) is valid.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Leak freedom</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ( { \phi } p { \text{emp} } ) is provable with <em>leak free</em> contracts then ( p ) is leak free.</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion
## Contributions

### Contracts

- Formalisation of contracts
- Automatic verification of contract properties

### Program analysis

- First extension of separation logic to message passing
- Formalisation of heap-manipulating, message passing programs with contracts
- Contracts and proofs collaborate to prove freedom from reception errors and leaks
- Tool that integrates this analysis: **Heap-Hop**
## Perspectives

### Contracts

- Prove progress for programs
- Extend to the multiparty case
- Enrich contracts with counters, non determinism, ...  

### Automatic program verification

- Discover specs and message footprints
- Discover contracts
- Fully automated tool
Program → Proof

Proof +

Contracts Prop. =

Program Prop. → SL+MP

Contracts

Heap-Hop