Heaps and Hops

Soutenance de thèse

Jules Villard

LSV, ENS Cachan, CNRS

Shift towards Concurrency

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


```
send(struct,e,data);
```

```
d = receive(struct,f);
```

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

Copyful


```
send(struct,e,data);
```

```
d = receive(struct,f);
```

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

Copyful


```
send(struct,e,data);
```

```
d = receive(struct,f);
```

Copyless


```
send(pointer,e,data);
```

```
d = receive(pointer,f);
```

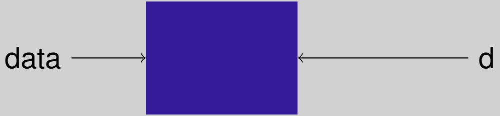
To Copy or not to Copy?

Copyful

`send(struct,e,data);`

`d = receive(struct,f);`

Copyless



`send(pointer,e,data);`

`d = receive(pointer,f);`

To Copy or not to Copy?

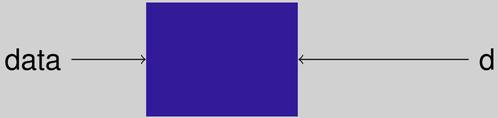
Copyful


```
send(struct,e,data);
```

```
d = receive(struct,f);
```

Copyless

Race!




```
send(pointer,e,data);
dispose(data);
```

```
d = receive(pointer,f);
dispose(d);
```

To Copy or not to Copy?

Copyful


```
send(struct,e,data);
```

```
d = receive(struct,f);
```

Copyless

Race!


```
send(pointer,e,data);
dispose(data);
```

```
d = receive(pointer,f);
dispose(d);
```

Copyful


```
send(struct,e,data);
```

```
d = receive(struct,f);
```

Copyless

No race

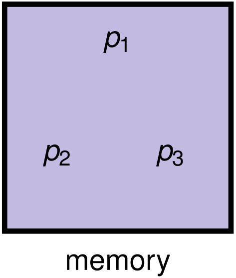

```
send(pointer,e,data);
```

```
d = receive(pointer,f);
dispose(d);
```

Singularity OS

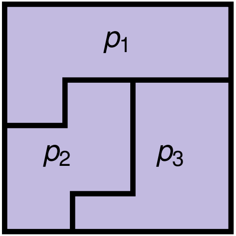
Singularity: a research project and an operating system.

- 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



memory

[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 π-calculus

- Communications are ruled by user-defined **contracts**

  similar to session types

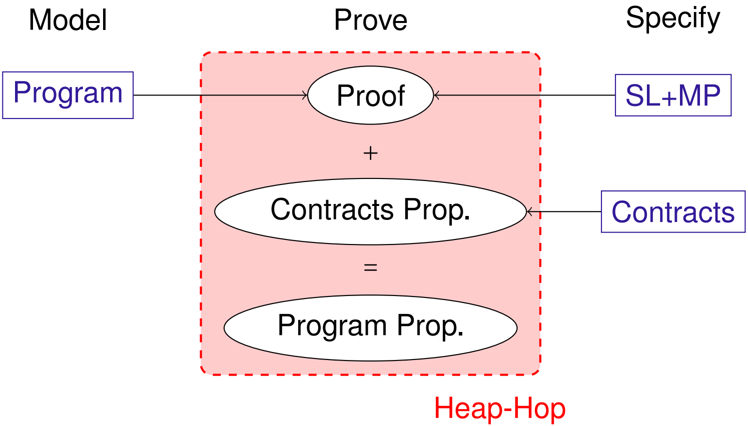
⊖ No formalisation

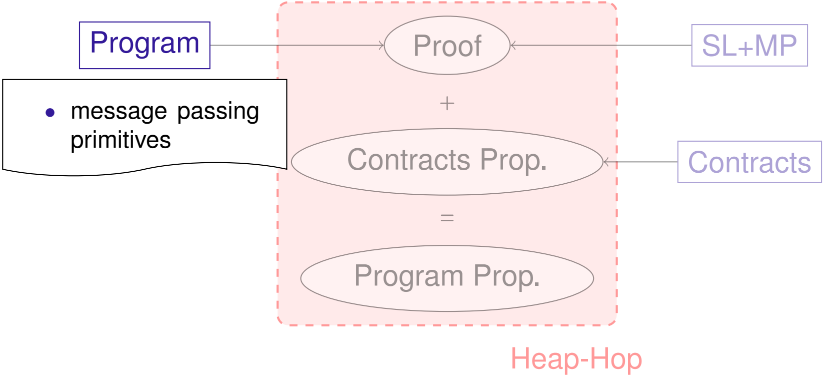
How to ensure the absence of bugs?

- **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


Model
Prove
Specify
Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop


Program
message passing primitives
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop

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

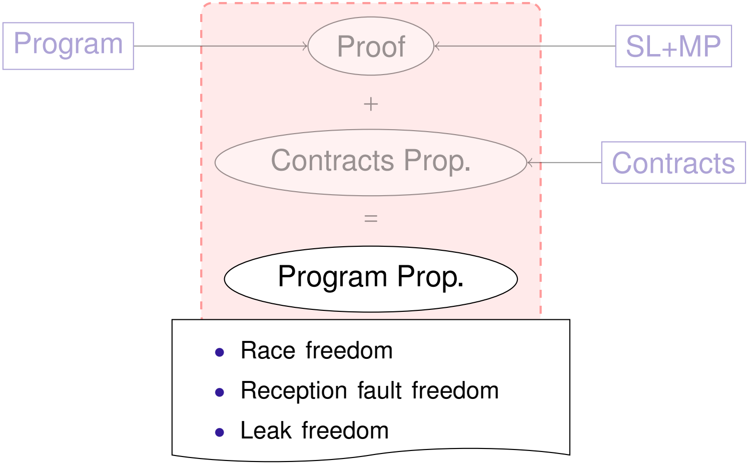
```
set_to_ten(x) {
  local e,f;
  (e,f) = open();
  send(integer,e,10);
  x = receive(integer,f);
  close(e,f);
}
```

Switch Receive

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

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

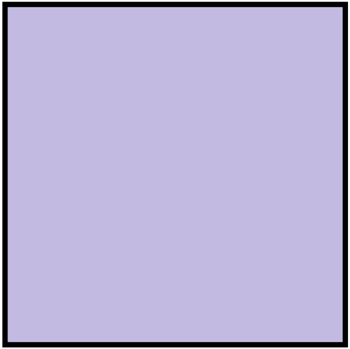
```
switch receive {
  y = receive(cell,f): {dispose(y);}
  z = receive(integer,f): {}
}
```


Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Race freedom
Reception fault freedom
Leak freedom

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.

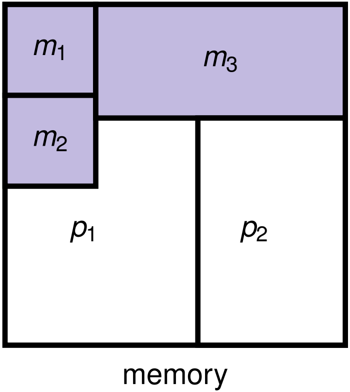


memory

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.

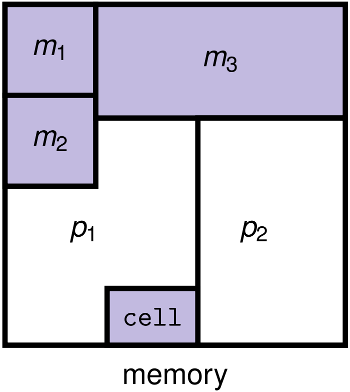
- Programs access only what they own.
- Prevents races.



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.

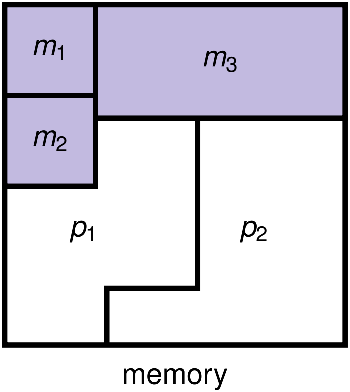
- Programs access only what they own.
- Prevents races.



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.



Separation property

Invalid receptions freedom

`switch receive` are exhaustive.

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

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

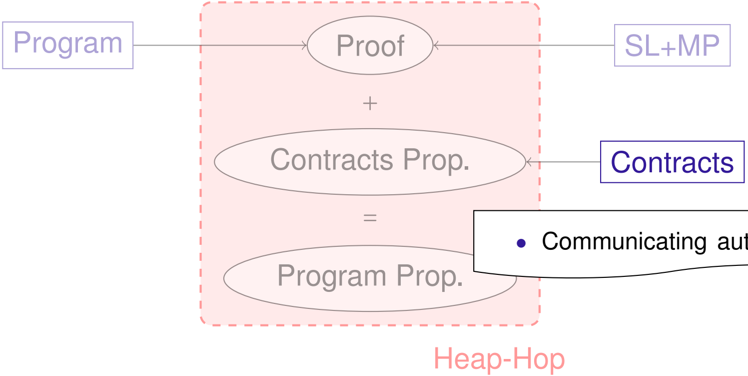
Separation property

Invalid receptions freedom

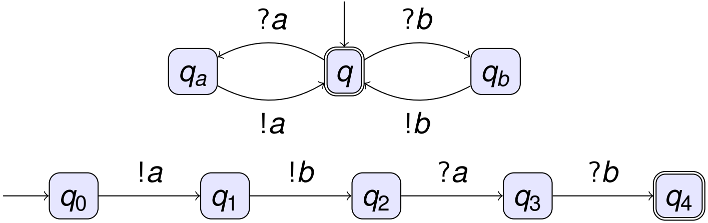
Leak freedom

The program does not leak memory.

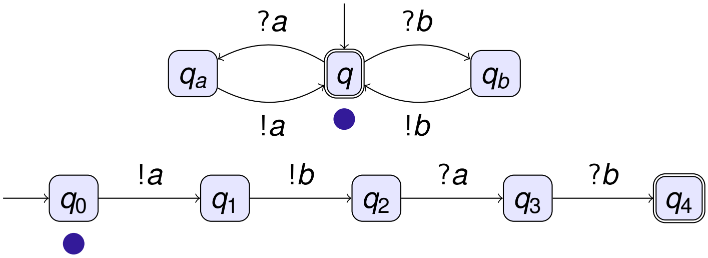
```
main() {
  local x,e,f;

  x = new();
  (e,f) = open();
  send(cell,e,x);
  close(e,f);
}
```



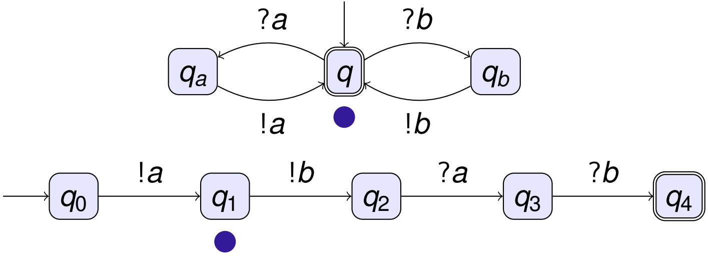
- Communicating automata



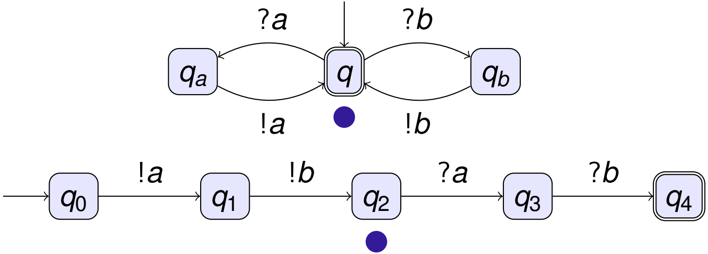
- Sending transitions: $!a$
- Receiving transitions: $?a$
- Two buffers: one in each direction
- Configuration: $\langle q, q', w, w' \rangle$



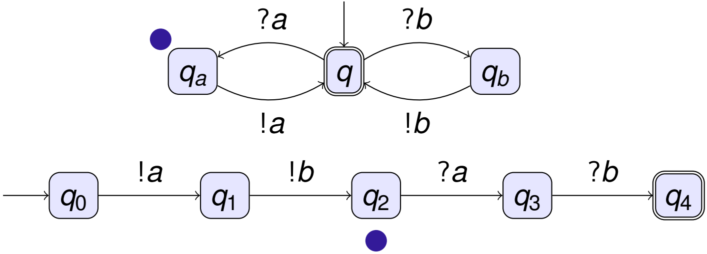
$$\langle q, q_0, \varepsilon, \varepsilon \rangle$$



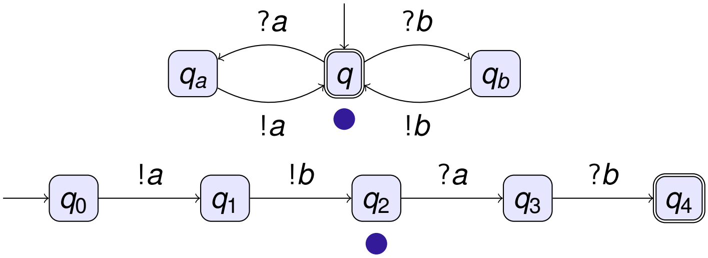
$$\langle q, q_1, a, \varepsilon \rangle$$



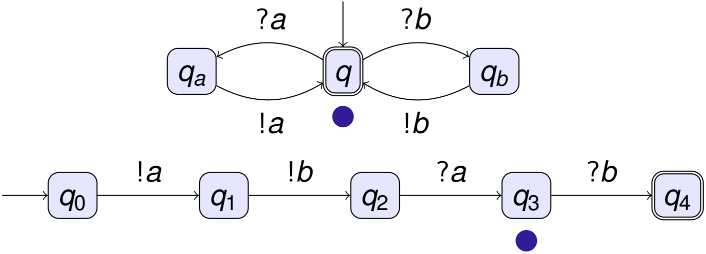
$$\langle q, q_2, ab, \varepsilon \rangle$$



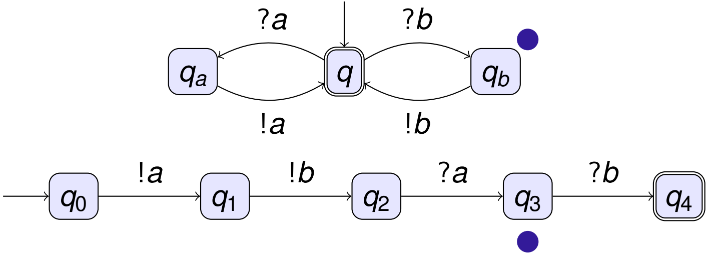
$$\langle q_a, q_2, b, \varepsilon \rangle$$



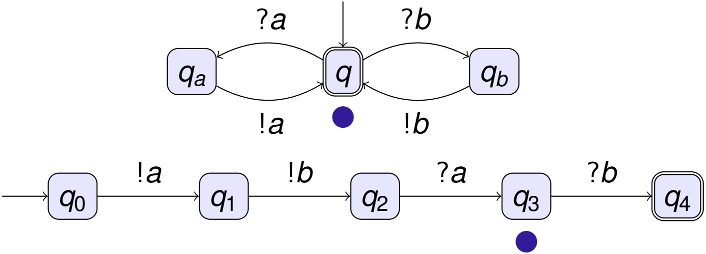
$$\langle q, q_2, b, a \rangle$$



$$\langle q, q_3, b, \varepsilon \rangle$$

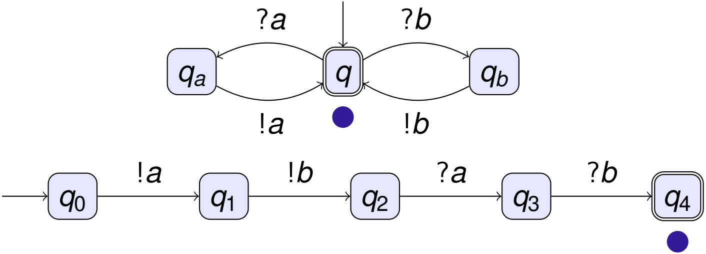


$$\langle q_b, q_3, \varepsilon, \varepsilon \rangle$$



$$\langle q, q_3, \varepsilon, b \rangle$$

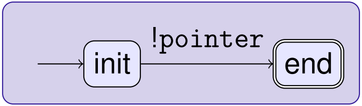
A Dialogue System



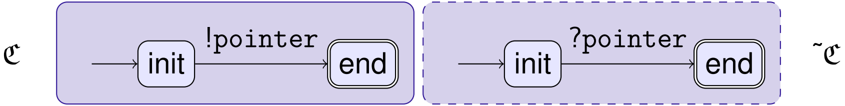
$$\langle q, q_4, \varepsilon, \varepsilon \rangle$$

Describe dual communicating finite state machines

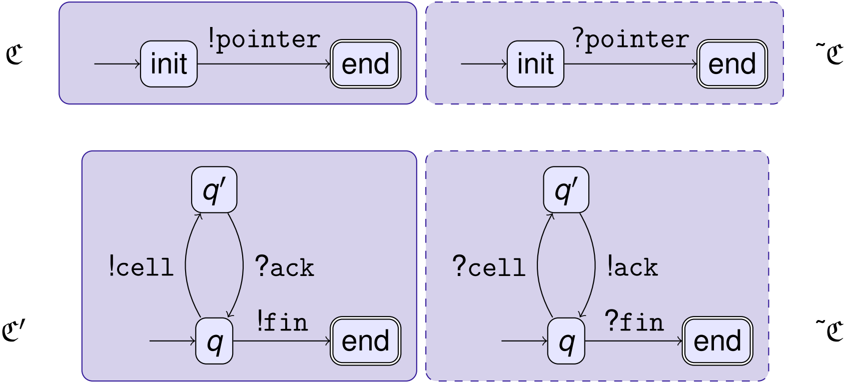
$\mathfrak{C}$



Describe dual communicating finite state machines

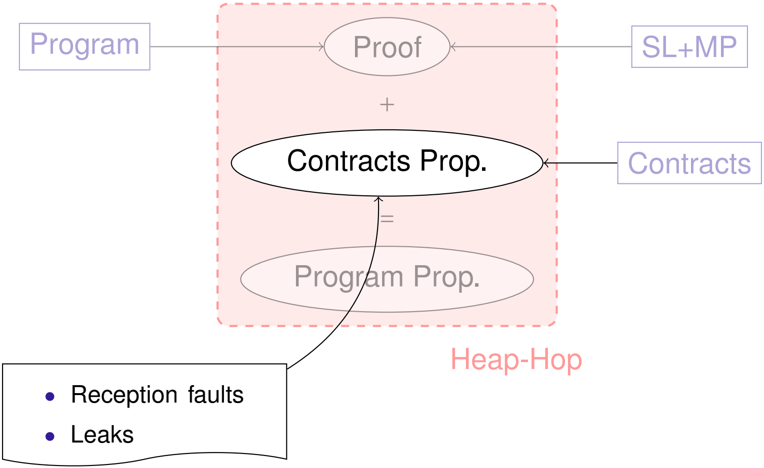


Describe dual communicating finite state machines



Contracts as Protocol Specifications

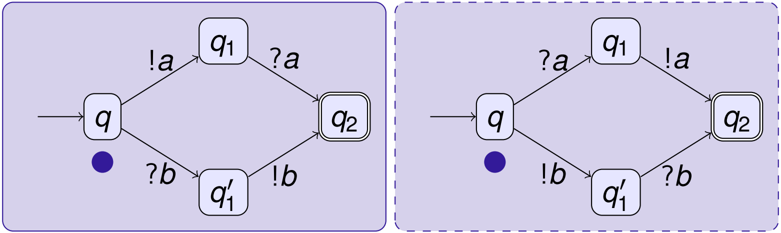
- `(e,f) = open(`$\mathfrak{C}$`)`: initialise endpoints in the initial state of the contract
- `send(a,e,x)`: becomes a $!a$ transition
- `y = receive(a,f)`: becomes a $?a$ transition
- `closed(e,f)` only when both endpoints are in the same **final** state.


Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop
Reception faults
Leaks

Definition — Reception fault

$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ 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 $b \neq a$

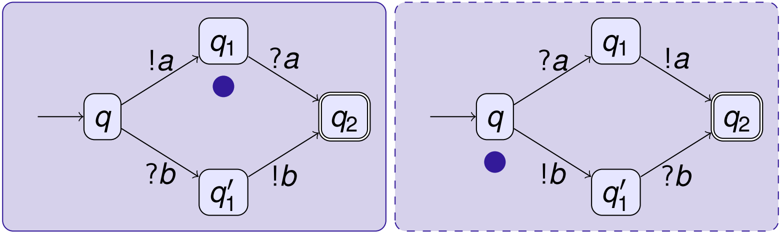


$\langle q, q, \varepsilon, \varepsilon \rangle$

Definition — Reception fault

$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ 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 $b \neq a$

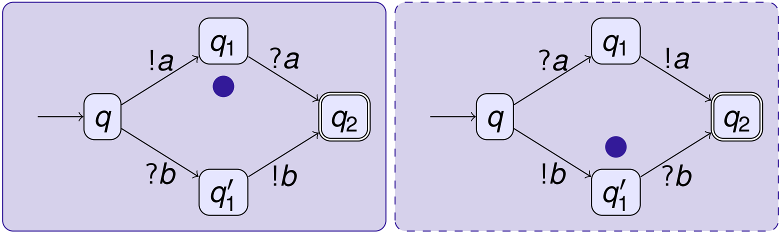


$$\langle q_1, q, a, \varepsilon \rangle$$

Definition — Reception fault

$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ 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 $b \neq a$

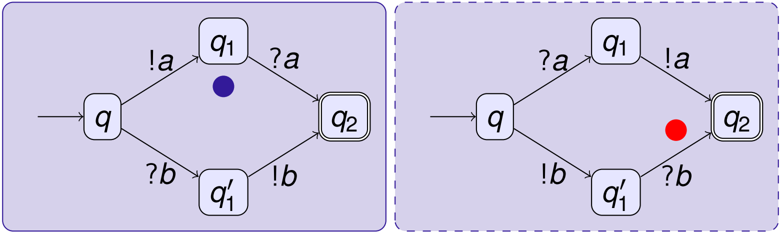


$$\langle q_1, q_1', a, b \rangle$$

Definition — Reception fault

$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ 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 $b \neq a$



$$\langle q_1, q'_1, a, b \rangle \xrightarrow{?b}_2 \textbf{error}$$

Definition — Reception fault

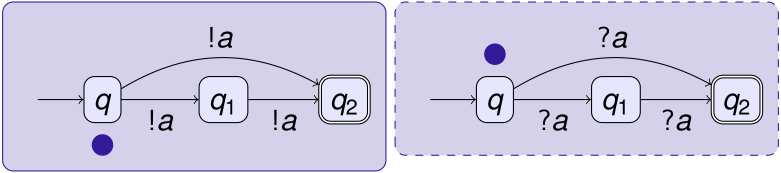
$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ is a **reception fault** if

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

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

Definition — Leak

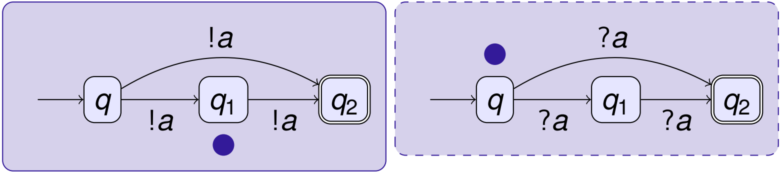
$\langle q_f, q_f, w_1, w_2\rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.



$$\langle q, q, \quad \varepsilon, \varepsilon\rangle$$

Definition — Leak

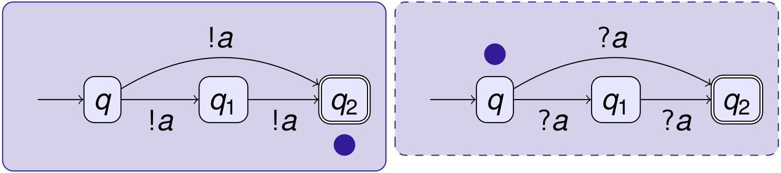
$\langle q_f, q_f, w_1, w_2 \rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.



$$\langle q_1, q, \quad a, \varepsilon \rangle$$

Definition — Leak

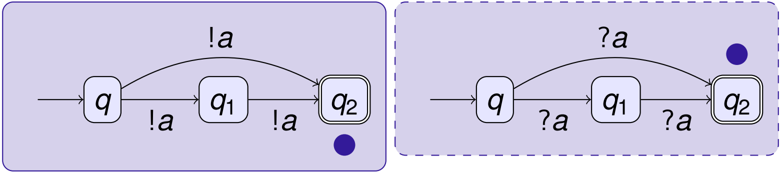
$\langle q_f, q_f, w_1, w_2 \rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.



$$\langle q_2, q, aa, \varepsilon \rangle$$

Definition Leak

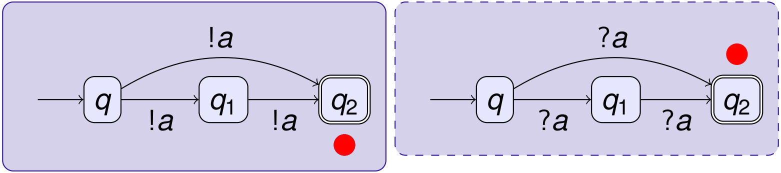
$\langle q_f, q_f, w_1, w_2 \rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.



$$\langle q_2, q_2, a, \varepsilon \rangle$$

Definition — Leak

$\langle q_f, q_f, w_1, w_2 \rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.



$\langle q_2, q_2, a, \varepsilon \rangle$

Undelivered Messages

Definition — Leak

$\langle q_f, q_f, w_1, w_2\rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.

- 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 ≈ 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 ≈ 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 ≈ Turing machine's tape
- Contracts are restricted (dual systems)
- ⊖ Contracts can encode Turing machines as well

Theorem

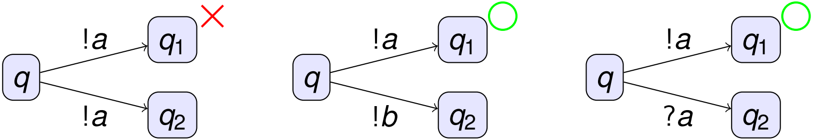
Safety is undecidable for contracts.

- 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.

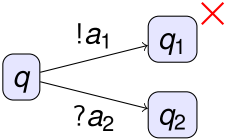


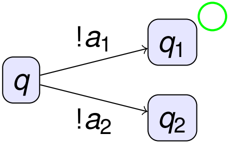
Sufficient Conditions for Reception Safety

Definition — Deterministic contract

Definition — Positional contracts

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





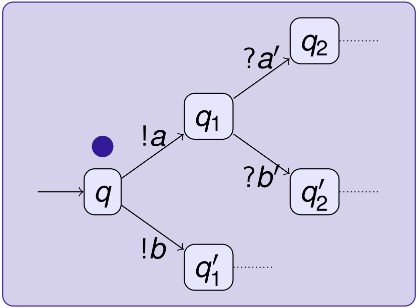
Sufficient Conditions for Reception Safety

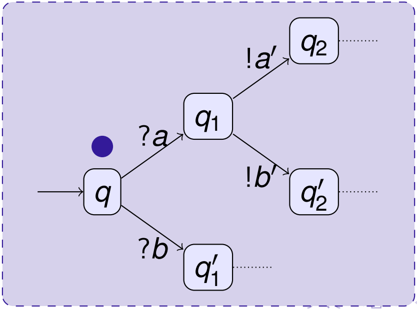
Definition — Deterministic contract

Definition — Positional contracts

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

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





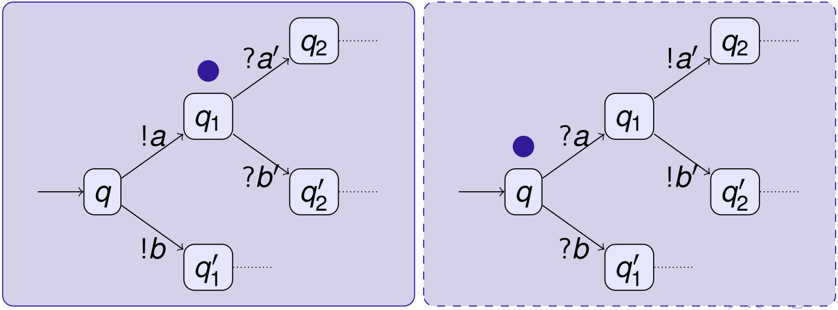
Sufficient Conditions for Reception Safety

Definition — Deterministic contract

Definition — Positional contracts

Theorem — [Stengel & Bultan'09] • [V., Lozes & Calcagno '09]

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



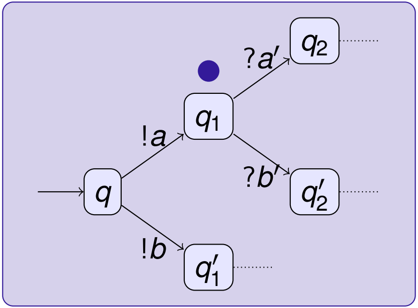
Sufficient Conditions for Reception Safety

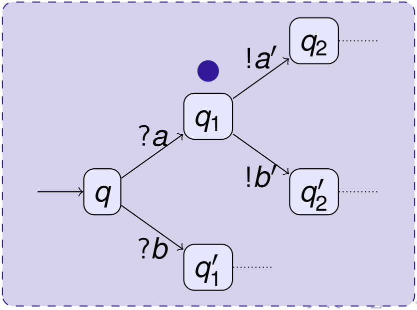
Definition Deterministic contract

Definition Positional contracts

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

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





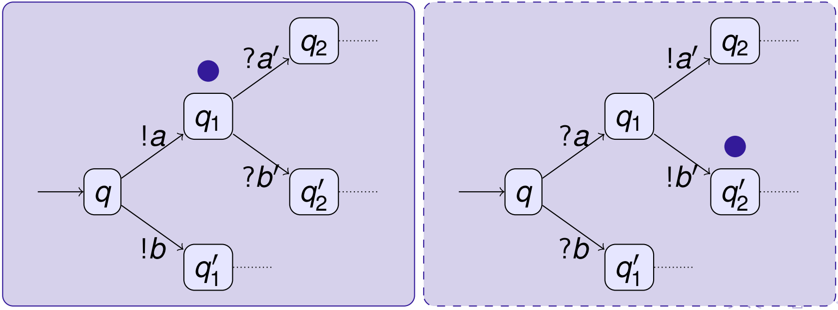
Sufficient Conditions for Reception Safety

Definition — Deterministic contract

Definition — Positional contracts

Theorem — [Stengel & Bultan'09] • [V., Lozes & Calcagno '09]

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



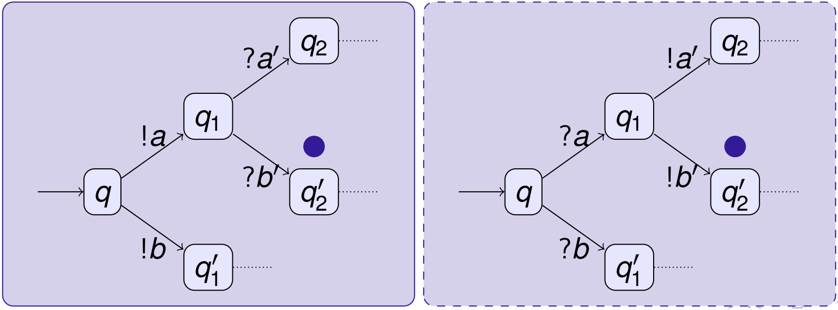
Sufficient Conditions for Reception Safety

Definition — Deterministic contract

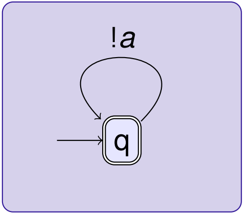
Definition — Positional contracts

Theorem — [Stengel & Bultan'09] • [V., Lozes & Calcagno '09]

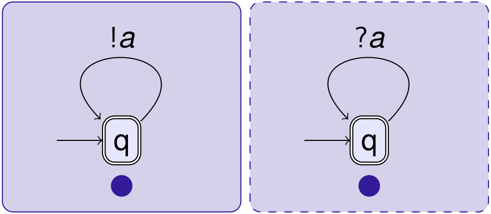
*Deterministic positional contracts are **reception fault free**.*



Another Source of Leaks

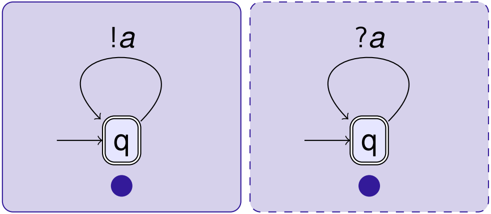


Another Source of Leaks



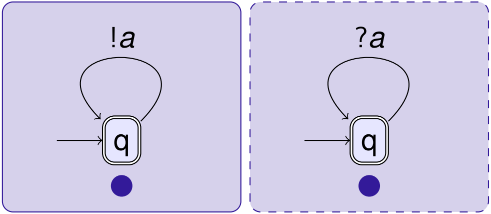
$$\langle q, q, \varepsilon, \varepsilon \rangle$$

Another Source of Leaks



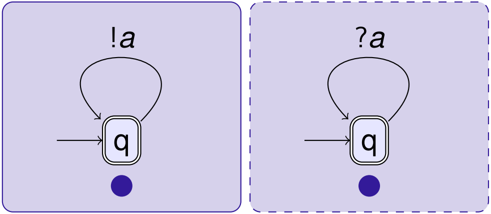
$\langle q, q, a, \varepsilon \rangle$

Another Source of Leaks



$$\langle q, q, aa, \varepsilon \rangle$$

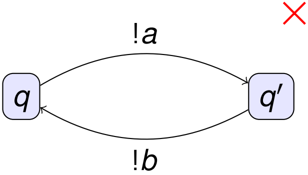
Another Source of Leaks

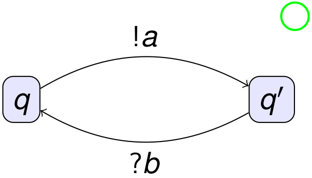


$$\langle q, q, aaa, \varepsilon\rangle$$

Definition — Synchronising state

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





Definition — Synchronising state

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

Definition — Synchronising contract

A contract is synchronising if all its final states are.

Synchronising Contracts

Definition — Synchronising state

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

Definition — Synchronising contract

A contract is synchronising if all its final states are.

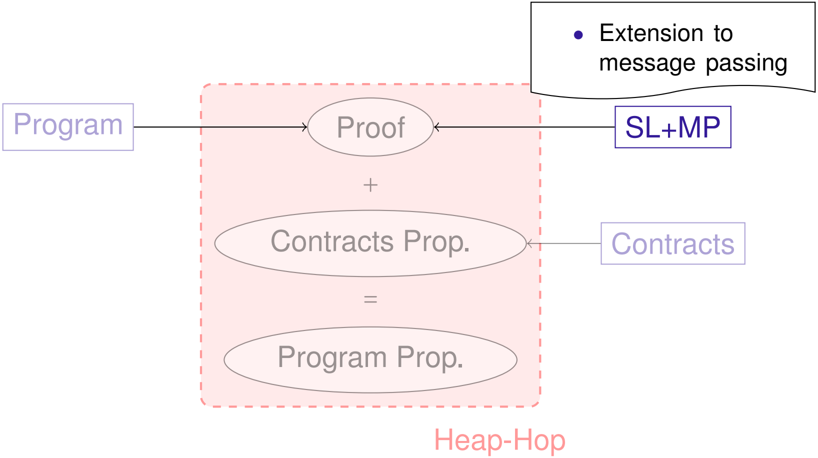
Theorem — [V., Lozes & Calcagno '09]

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

Definition — Singularity contract

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**.



Extension to message passing
Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop

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

- An **assertion language** to describe states
- A **proof system** for Hoare triples

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

Syntax

E	$::=$	$x \mid n \in \{0, 1, 2, \ldots\} \mid \cdots$	expressions
ϕ	$::=$	$E_1 = E_2 \mid E_1 \neq E_2$	stack predicates
		$\mid \mathrm{emp} \mid E_1 \mapsto E_2$	heap predicates
		$\mid \exists x.\, \phi \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 * \phi_2$	formulas

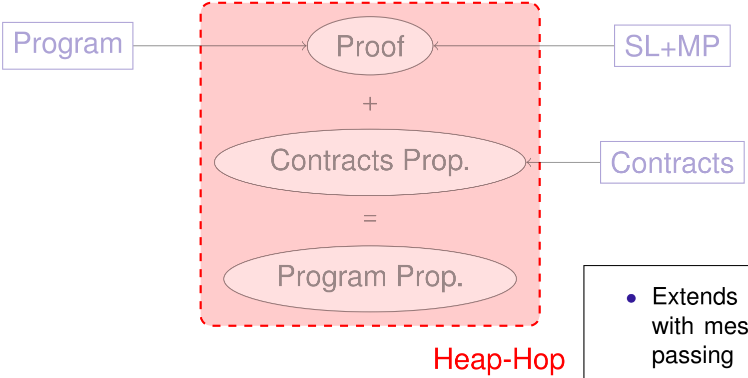
Assertion Language (extension)

Syntax (continued)

$$\phi ::= \ldots \mid E \mapsto (\mathfrak{C}\{q\}, E') \qquad \text{endpoint predicate}$$

Intuitively $E \mapsto (\mathfrak{C}\{q\}, E')$ means:

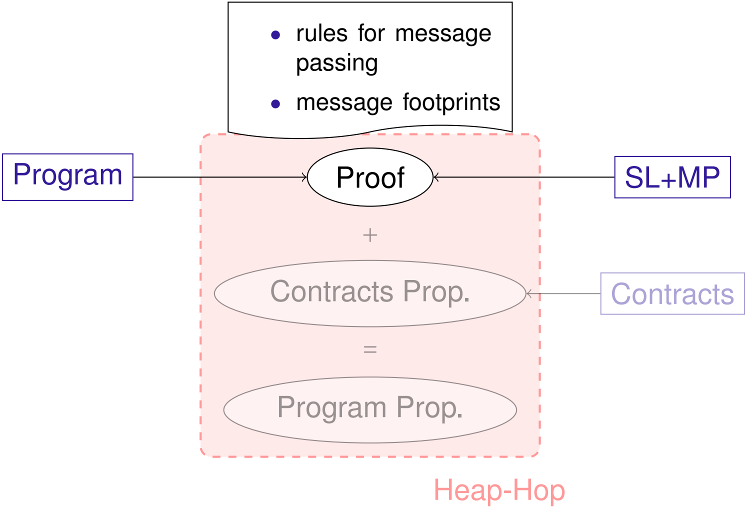
- E is an allocated endpoint
- it is ruled by contract $\mathfrak{C}$
- it is currently in the control state q of $\mathfrak{C}$
- its peer is E'



- Extends Smallfoot with message passing
- Written in OCaml
- Open source

HEAP HOP

[V., Lozes & Calcagno TACAS'10]



rules for message passing
message footprints
Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop

Memory States σ

A memory state σ 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:

$$p, \sigma \to^* p', \sigma' \qquad \text{(intermediate state)}$$
$$p, \sigma \to^* \sigma' \qquad \text{(final state)}$$
$$p, \sigma \to^* \textbf{error} \qquad \text{(error state)}$$

$\{\phi\}$ *p* $\{\psi\}$: Hoare triple

- ϕ, ψ: formulas
- *p*: program

Fault-free interpretation of Hoare triples

If $\{\phi\}$ *p* $\{\psi\}$ is provable, then for all state $\sigma \vDash \phi$,

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

Proof system

Derivation rules to **prove** Hoare triples.

Rules of the Proof System

SKIP	ASSUME	ASSIGN	LOOKUP	MUTATE
NEW	DISPOSE	SEQUENCE	PARALLEL	CHOICE
STAR	LOCAL	FRAME	WEAKENING	
CONJUNCTION	DISJUNCTION	EXISTENTIAL	OPEN	
CLOSE	SEND	CHANNELDISPATCH	EXTCHOICE	

Communication Rules

Open

$$\frac{i = \mathrm{init}(\mathfrak{C})}{\{\mathrm{emp}\}\ (\mathrm{e},\mathrm{f}) = \mathrm{open}(\mathfrak{C})\ \{e \mapsto (\mathfrak{C}\{i\}, f) * f \mapsto (\mathfrak{C}\{i\}, e)\}}$$

Close

$$\frac{q \in \mathrm{finals}(\mathfrak{C})}{\{e \mapsto (\mathfrak{C}\{q\}, f) * f \mapsto (\tilde{\mathfrak{C}}\{q\}, e)\}\ \mathrm{close}(\mathrm{e},\mathrm{f})\ \{\mathrm{emp}\}}$$

Send

$$\frac{q \xrightarrow{!a} q' \in \mathfrak{C} \qquad e \mapsto (\mathfrak{C}\{q'\}, -) * \phi \Rightarrow \gamma_a(e, x) * \phi'}{\{e \mapsto (\mathfrak{C}\{q\}, -) * \phi\}\ \mathrm{send}(\mathrm{a},\mathrm{e},\mathrm{x})\ \{\phi'\}}$$

Receive

$$\frac{q \xrightarrow{?a} q' \in \mathfrak{C}}{\{e \mapsto (\mathfrak{C}\{q\}, X')\}\ \mathrm{x} = \mathrm{receive}(\mathrm{a},\mathrm{e})\ \{e \mapsto (\mathfrak{C}\{q'\}, X') * \gamma_a(X', x)\}}$$

Communication Rules

CLOSE

$$\frac{q \in \text{finals}(\mathfrak{C})}{\{e \mapsto (\mathfrak{C}\{q\}, f) * f \mapsto (\tilde{\mathfrak{C}}\{q\}, e)\}\ \text{close}(\text{e},\text{f})\ \{\text{emp}\}}$$

RECEIVE

$$\frac{q \xrightarrow{?a} q' \in \mathfrak{C}}{\{e \mapsto (\mathfrak{C}\{q\}, X')\}\ \text{x} = \text{receive}(\text{a},\text{e})\ \{e \mapsto (\mathfrak{C}\{q'\}, X') * \gamma_a(X', x)\}}$$

Closing a Channel

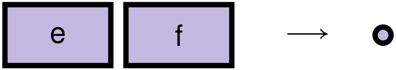
CLOSE

$$\frac{q \in \mathsf{finals}(\mathfrak{C})}{\{e \mapsto (\mathfrak{C}\{q\}, f) * f \mapsto (\bar{\mathfrak{C}}\{q\}, e)\}\ \mathsf{close}(\mathsf{e},\mathsf{f})\ \{\mathsf{emp}\}}$$

Closing a Channel

CLOSE

$$\frac{q \in \mathsf{finals}(\mathfrak{C})}{\{e \mapsto (\mathfrak{C}\{q\}, f) * f \mapsto (\tilde{\mathfrak{C}}\{q\}, e)\}\ \mathsf{close}(e,f)\ \{\mathsf{emp}\}}$$



General Rule for Receive

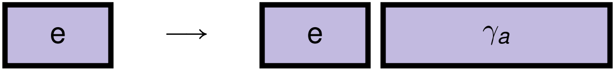
Receive

$$\frac{q \xrightarrow{?a} q' \in \mathfrak{C}}{\{e \mapsto (\mathfrak{C}\{q\}, X')\}\ \texttt{x = receive(a,e)}\ \{e \mapsto (\mathfrak{C}\{q'\}, X') * \gamma_a(X', x)\}}$$

General Rule for Receive

RECEIVE

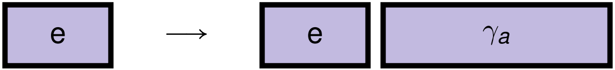
$$\frac{q \xrightarrow{?a} q' \in \mathfrak{C}}{\{e \mapsto (\mathfrak{C}\{q\}, X')\}\ \texttt{x = receive(a,e)}\ \{e \mapsto (\mathfrak{C}\{q'\}, X') * \gamma_a(X', x)\}}$$



General Rule for Receive

RECEIVE

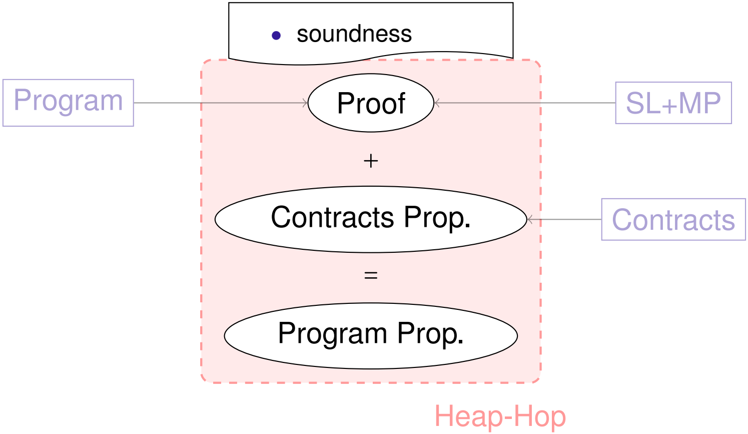
$$\frac{q \xrightarrow{?a} q' \in \mathfrak{C}}{\{e \mapsto (\mathfrak{C}\{q\}, X')\}\ \texttt{x = receive(a,e)}\ \{e \mapsto (\mathfrak{C}\{q'\}, X') * \gamma_a(X', x)\}}$$



Can be instantiated for each example:

$$\gamma_{\text{cell}}(\text{src}, \text{val}) \triangleq \text{val} \mapsto -$$

$$\gamma_{\text{ep}}(\text{src}, \text{val}) \triangleq \text{val} \mapsto (\mathfrak{C}\{\text{end}\}, -) \land \text{val} = \text{src}$$


• soundness
Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop

Validity and Leak Freedom

Definition — Program validity

$\{\phi\}\ p\ \{\psi\}$ is valid if, for all $\sigma \vDash \phi$

- p has **no race or memory fault** starting from σ
- p has **no reception faults** starting from σ
- if $p, \sigma \rightarrow^* \sigma'$ then $\sigma' \vDash \psi$

Definition — Leak free programs

p is **leak free** if for all σ

$p, \sigma \rightarrow^* \sigma'$ implies that the heap and buffers of σ' are empty

Properties of Proved Programs

Theorem — Soundness

*If $\{\phi\}$ p $\{\psi\}$ is provable with **reception fault free** contracts then $\{\phi\}$ p $\{\psi\}$ is valid.*

Theorem — Leak freedom

*If $\{\phi\}$ p $\{\text{emp}\}$ is provable with **leak free** contracts then p is leak free.*

Conclusion

Contracts

- Formalisation of contracts
- Automatic verification of contract properties

Program analysis

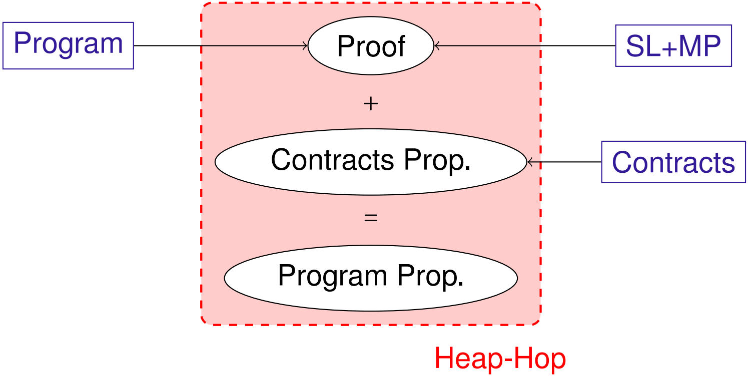
- First extension of separation logic to message passing
- Formalisation of heap-manipulating, message passing programs with contracts
- Contracts and proofs collaborate to prove freedom from reception errors and leaks
- Tool that integrates this analysis: **Heap-Hop**

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
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop