Tracking Heaps that Hop with Heap-Hop

Jules Villard[1,3] Étienne Lozes[2,3] Cristiano Calcagno[4,5]

[1]Queen Mary, University of London

[2]RWTH Aachen, Germany

[3]LSV, ENS Cachan, CNRS

[4]Monoidics, Inc.

[5]Imperial College, London

Message Passing in Multicore Systems

- Hard to write sequential programs that are both correct and efficient

Message Passing in Multicore Systems

- Hard to write sequential programs that are both correct and efficient
- Hard to write concurrent programs that are both/either correct and/or efficient

Message Passing in Multicore Systems

- Hard to write sequential programs that are both correct and efficient
- Hard to write concurrent programs that are both/either correct and/or efficient
- Paradigm: message passing over a shared memory
- Leads to **efficient**, copyless message passing
- May be more error-prone (than message passing with copies)

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

To Copy or not to Copy?

Copyful

`send(struct,e,data);`

`d = receive(struct,f);`

Copyless

`send(pointer,e,data);`

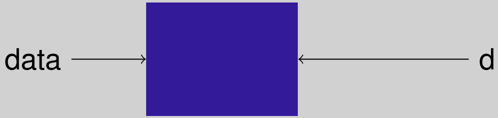
`d = receive(pointer,f);`

Copyful


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

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

Copyless




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

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

To Copy or not to Copy?

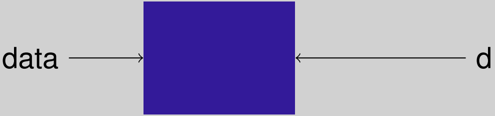
Copyful


```
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

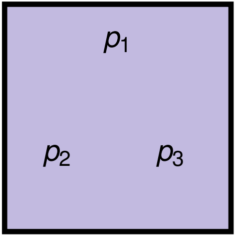
No race


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

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

Singularity: a research project and an operating system.

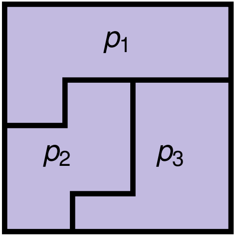
- No hardware memory protection
- Sing♯ language
- Isolation is verified at compile time
- Invariant: each memory cell is owned by at most one thread
- No shared resources
- Copyless message passing



memory

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

Singularity Channels [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

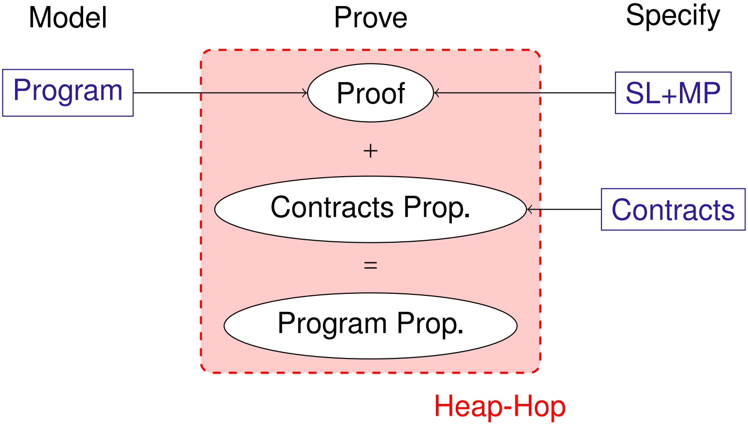
  similar to the π-calculus
- Communications are ruled by user-defined **contracts**

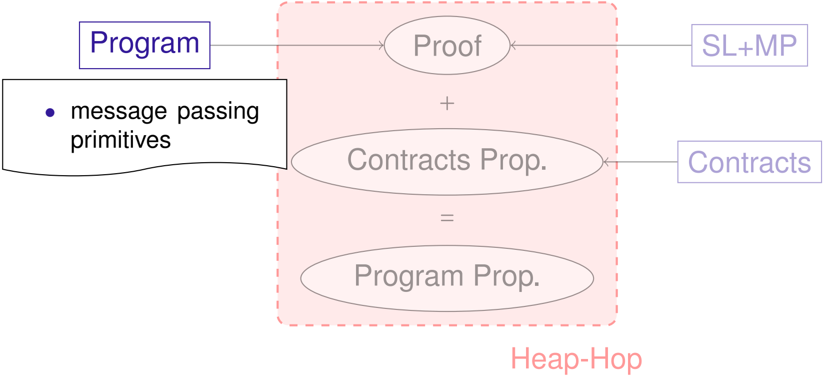
  similar to session types

⊖ No formalisation

How to ensure the absence of bugs?

Analysis [V., Lozes & Calcagno APLAS'09,V. PhD'11]




Program
message passing primitives
Proof
+
Contracts Prop.
=
Program Prop.
SL+MP
Contracts
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`

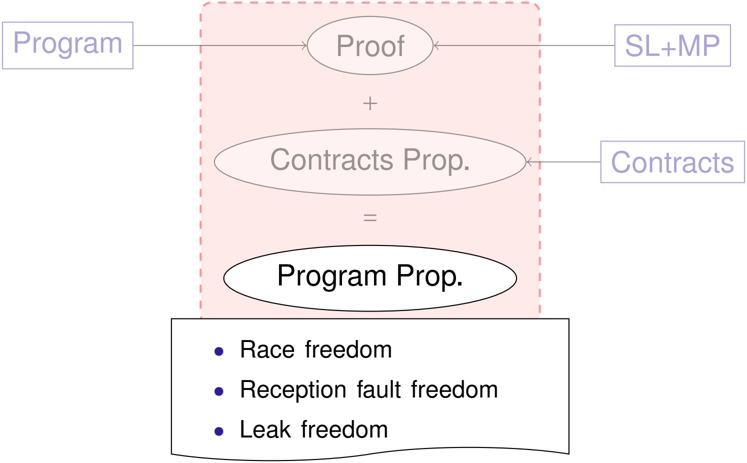
```
set_to_ten(x) {
  local e,f;
  (e,f) = open();
  send(integer,e,10);
  x = receive(integer,f);
  close(e,f);
}
```

Switch Receive

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

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

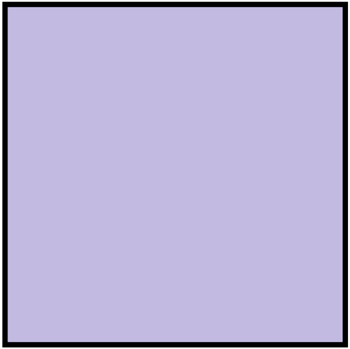
```
switch receive {
  y = receive(cell,f): {dispose(y);}
  z = receive(integer,f): {}
}
```


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
- Linear usage of channels

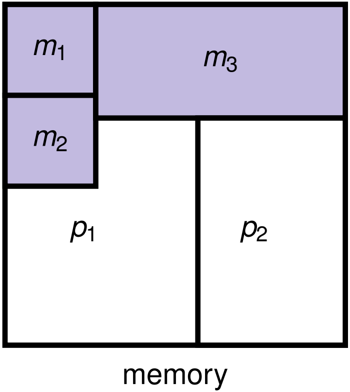


memory

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.

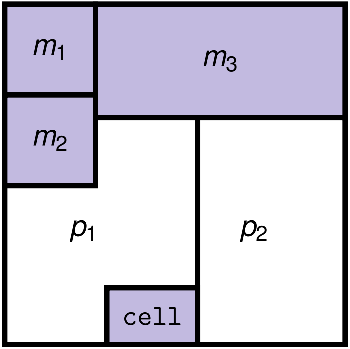
- Programs access only what they own
- Prevents races
- Linear usage of channels



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
- Linear usage of channels

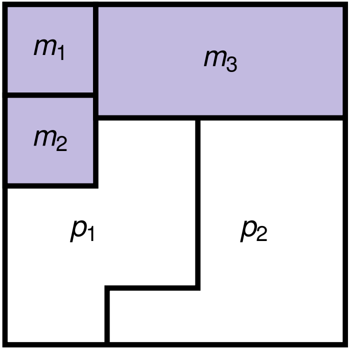


memory

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
- Linear usage of channels



memory

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

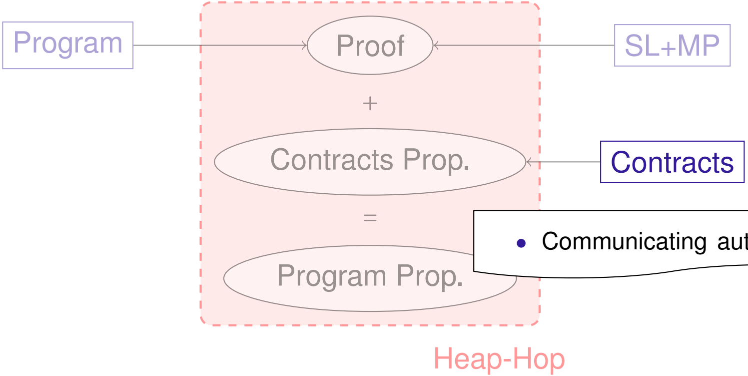
Invalid receptions freedom

Leak freedom

The program does not leak memory.

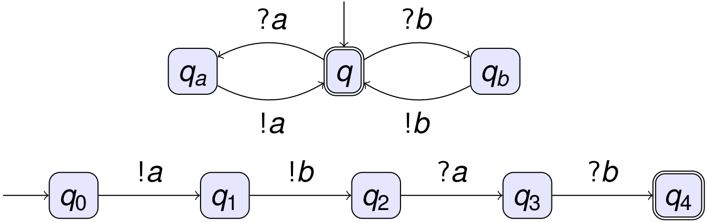
```
main() {
  local x,e,f;

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

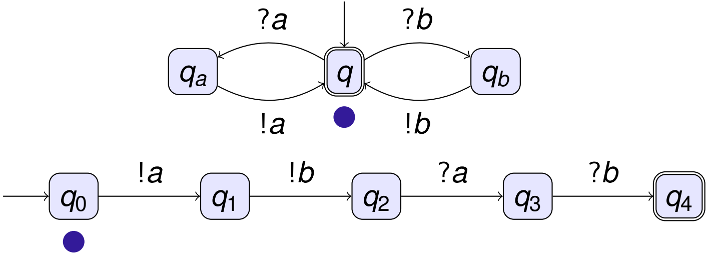


- Communicating automata

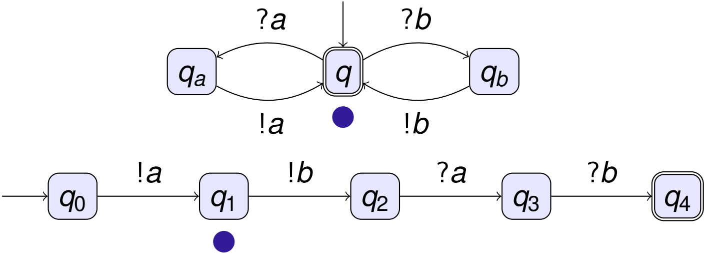
A Dialogue System



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

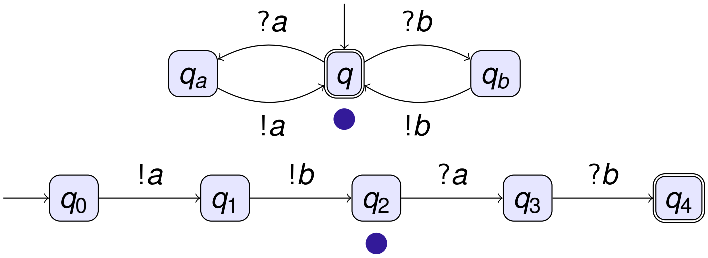


$$\langle q, q_0, \varepsilon, \varepsilon \rangle$$

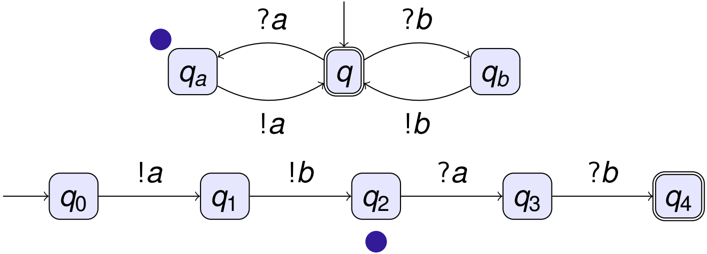


$$\langle q, q_1, a, \varepsilon \rangle$$

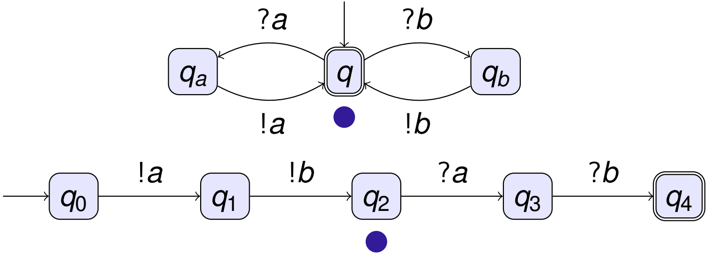
A Dialogue System



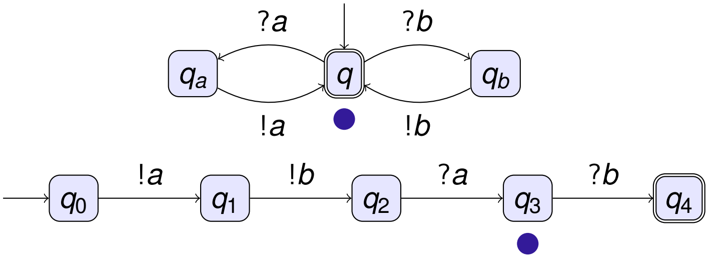
$$\langle q, q_2, ab, \varepsilon \rangle$$



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

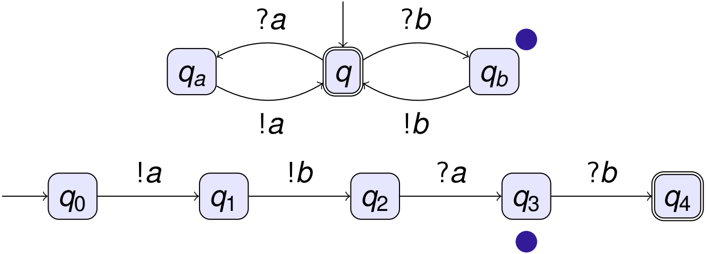


$$\langle q, q_2, b, a \rangle$$

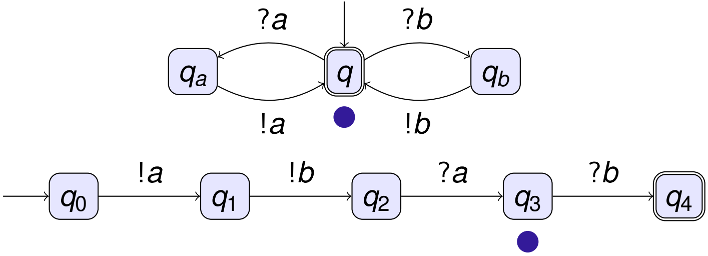


$$\langle q, q_3, b, \varepsilon \rangle$$

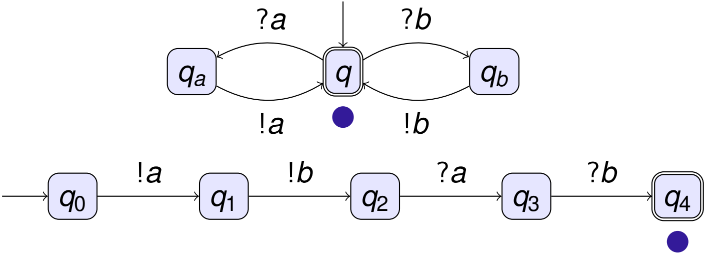
A Dialogue System



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



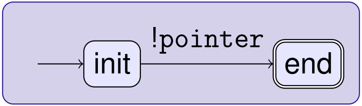
$$\langle q, q_3, \varepsilon, b \rangle$$



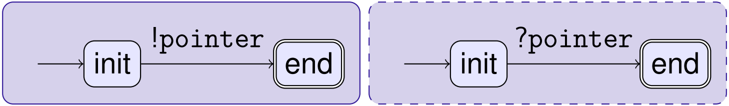
$$\langle q, q_4, \varepsilon, \varepsilon \rangle$$

Describe dual communicating finite state machines

$\mathfrak{C}$

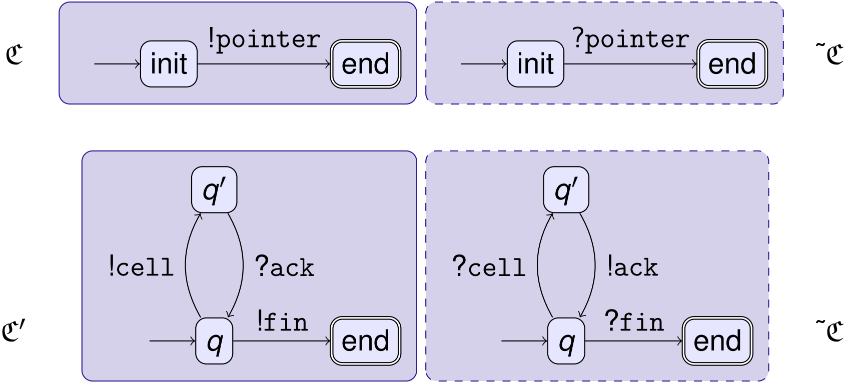


Describe dual communicating finite state machines

$\mathfrak{C}$  $\tilde{\mathfrak{C}}$

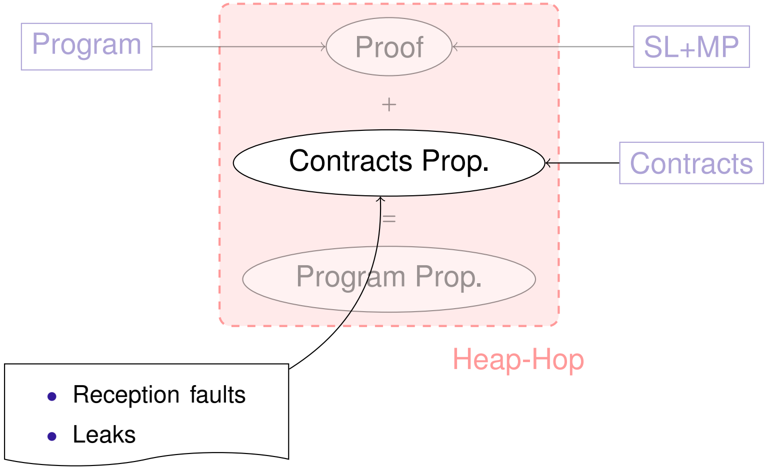
Contracts

Describe dual communicating finite state machines



Contracts as Protocol Specifications

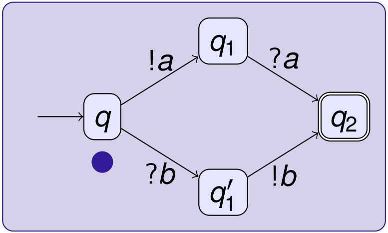
- `(e,f) = open(`$\mathfrak{C}$`)`: initialise endpoints in the initial state of the contract
- `send(a,e,x)`: becomes a $!a$ transition
- `y = receive(a,f)`: becomes a $?a$ transition
- `close(e,f)` only when both endpoints are in the same **final** state.

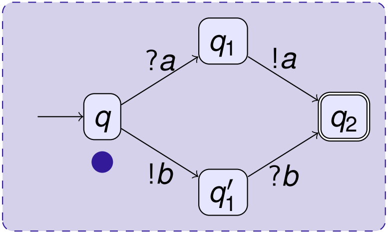

Program
Proof
SL+MP
+
Contracts Prop.
Contracts
=
Program Prop.
Heap-Hop
• Reception faults
• Leaks

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 $b \neq a$



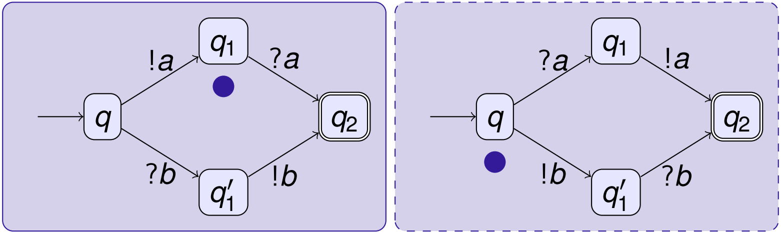


$\langle q, q, \varepsilon, \varepsilon \rangle$

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 $b \neq a$

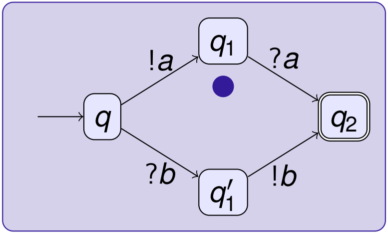


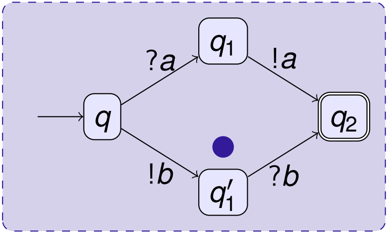
$\langle q_1, q, a, \varepsilon \rangle$

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 $b \neq a$



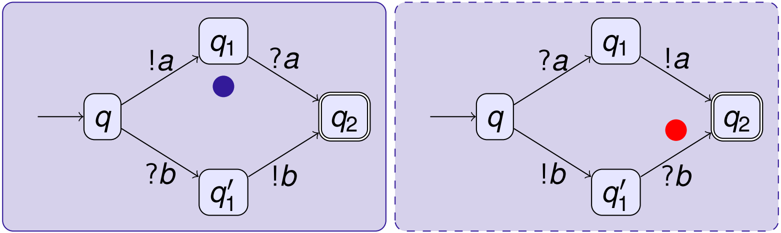


$\langle q_1, q'_1, a, b \rangle$

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

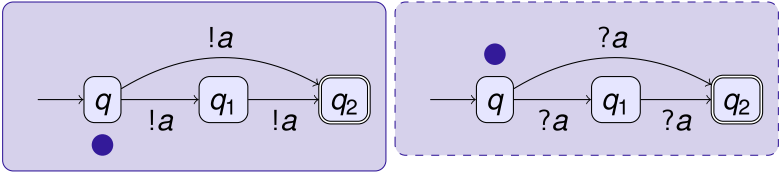
$\langle q_1, q_2, a \cdot w_1, w_2 \rangle$ is a **reception fault** if

- $q_1 \xrightarrow{?b} q$ 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

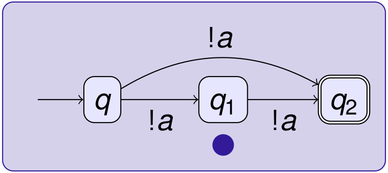
$\langle q_f, q_f, w_1, w_2 \rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.

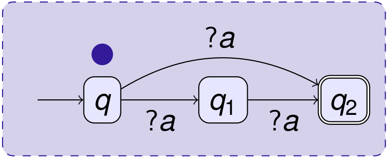


$$\langle q, q, \quad \varepsilon, \varepsilon \rangle$$

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.

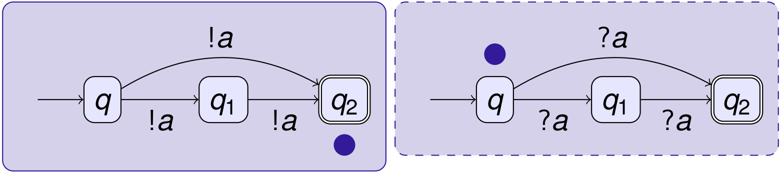




$$\langle q_1, q, \quad a, \varepsilon\rangle$$

Definition — **Leak**

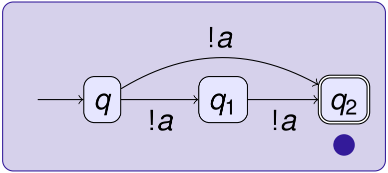
$\langle q_f, q_f, w_1, w_2\rangle$ is a **leak** if $w_1 \cdot w_2 \neq \varepsilon$ and q_f is final.

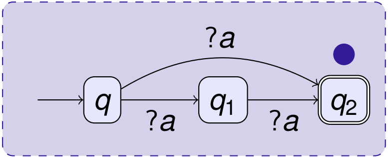


$$\langle q_2, q, aa, \varepsilon\rangle$$

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.

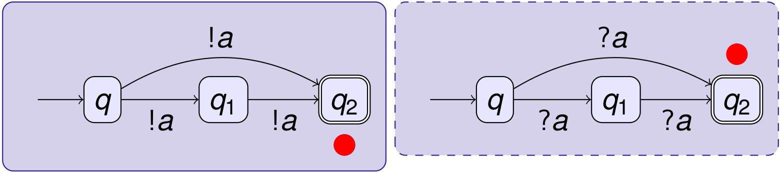




$$\langle q_2, q_2, a, \varepsilon \rangle$$

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

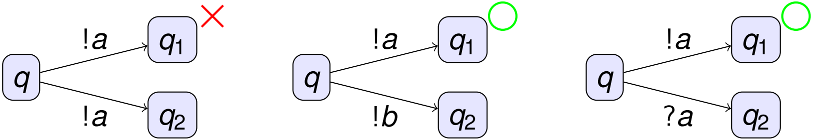
Safety is undecidable for contracts.

- 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.

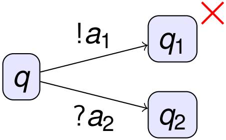


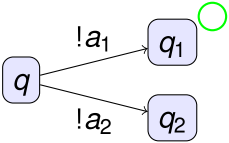
Sufficient Conditions for Reception Safety

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

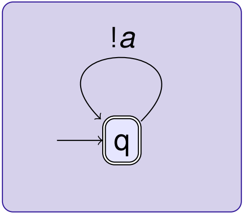
Definition — Deterministic contract

Definition — Positional contracts

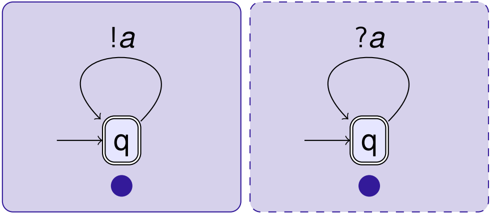
Theorem [Stengel & Bultan'09] • [V., Lozes & Calcagno '09]

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

Another Source of Leaks

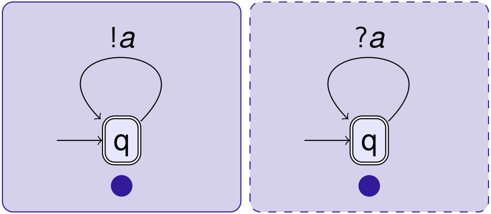


Another Source of Leaks



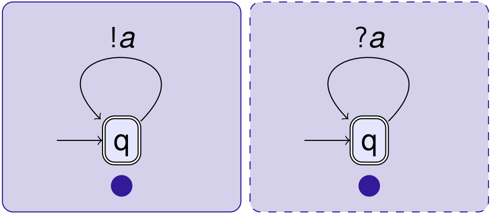
$$\langle q, q, \varepsilon, \varepsilon \rangle$$

Another Source of Leaks



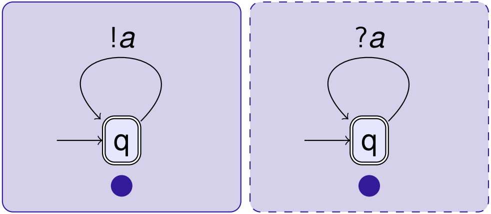
$$\langle q, q, a, \varepsilon \rangle$$

Another Source of Leaks



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

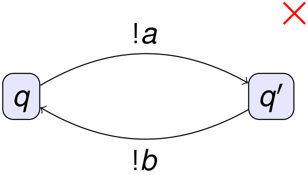
Another Source of Leaks

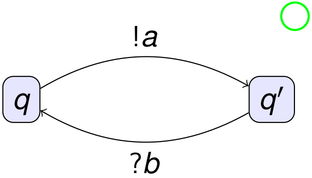


$$\langle q, q, aaa, \varepsilon \rangle$$

Definition — **Synchronising state**

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





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.

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.

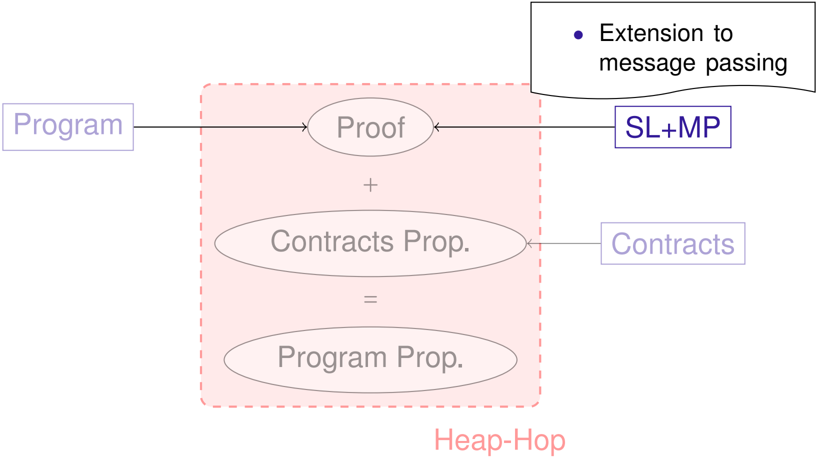
Theorem — [V., Lozes & Calcagno '09]

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, ...]

- Local reasoning for heap-manipulating programs
- Naturally describes ownership transfers
- Numerous extensions, *e.g.* storable locks [Gotsman et al. 07]

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

- Local reasoning for heap-manipulating programs
- Naturally describes ownership transfers
- Numerous extensions, *e.g.* storable locks [Gotsman et al. 07]

New Now with message passing! [APLAS'09]

Syntax of SL

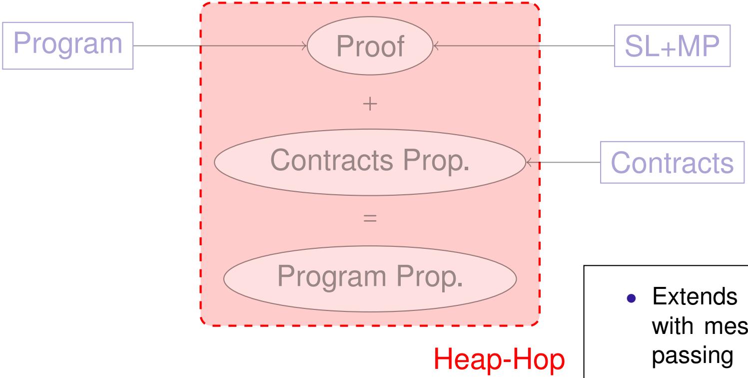
$$
\begin{array}{lcll}
E & ::= & x \mid n \in \{0, 1, 2, \ldots\} \mid \cdots & \text{expressions} \\
\phi & ::= & E_1 = E_2 \mid E_1 \neq E_2 & \text{stack predicates} \\
 & & \mid \mathrm{emp} \mid E_1 \mapsto E_2 & \text{heap predicates} \\
 & & \mid \exists x.\, \phi \mid \phi_1 \wedge \phi_2 \mid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

Syntax (continued)

$$\begin{array}{lll} \phi & ::= & \dots \\ & & \mid E \mapsto (\mathfrak{C}\{q\}, E') \qquad \text{endpoint predicate} \end{array}$$

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

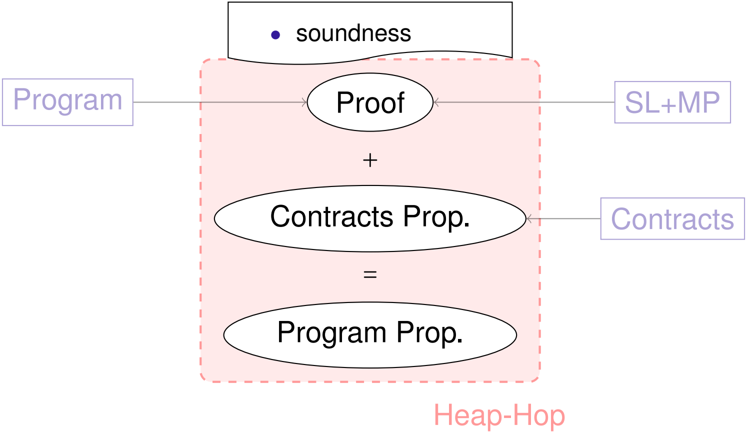
- E is an allocated endpoint
- it is ruled by contract $\mathfrak{C}$
- it is currently in the control state q of $\mathfrak{C}$
- its peer is E'



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

HEAP HOP

[V., Lozes & Calcagno TACAS'10]



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

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 $\{\mathrm{emp}\}$ is provable with **leak free** contracts then p is **leak free**.*

Conclusion

Contributions

Contracts

- Formalisation of contracts
- Automatic verification of contract properties

Program analysis

- Verification 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 (counters, non positional, ...)

Today@5:15 More general property of contracts for decidability: half-duplex

Automatic program verification

- Discover specs and message footprints
- Discover contracts
- Fully automated tool

Tracking Heaps that Hop with Heap-Hop

Jules Villard[1,3] Étienne Lozes[2,3] Cristiano Calcagno[4,5]

[1]Queen Mary, University of London

[2]RWTH Aachen, Germany

[3]LSV, ENS Cachan, CNRS

[4]Monoidics, Inc.

[5]Imperial College, London