



COLLÈGE  
DE FRANCE  
—1530—

# Some extensions of separation logic

---

Xavier Leroy

Collège de France, chair of software sciences

`xavier.leroy@college-de-france.fr`

**Waving the magic wand**



## An adaptation problem

Often, in a verification step, we want to apply

- a “small rule”  $\{ \ell \mapsto - \} \text{set}(\ell, v) \{ \lambda.. \ell \mapsto v \}$
- or a “small specification” for a function  $\{ \text{list}(w, p) \} \text{reverse}(p) \{ \lambda r. \text{list}(\text{rev}(w), r) \}$

in a bigger context, such as

$$\text{list}(p, w) * \text{list}(q, w') * \langle x > 0 \rangle * t \mapsto x * t + 1 \mapsto q$$

In general, we need to 1- unroll representation predicates,  
2- find a framing, 3- apply the consequence rule.

## The framed consequence rule

(Derived from the frame rule + the consequence rule.)

A general ways to adapt what we already know  $\{P'\} c \{Q'\}$  to what we need to prove  $\{P\} c \{Q\}$ .

$$\frac{\{P'\} c \{Q'\} \quad P \Rightarrow P' \star R \quad \forall v, Q' v \star R \Rightarrow Q v}{\{P\} c \{Q\}}$$

Automated or semi-automated theorem proving works well to show the implications  $P \Rightarrow P' \star R$  and  $Q' v \star R \Rightarrow Q v$ .

The difficulty is to find assertion  $R$ .

# Abduction and magic wand

## The abduction problem

Given  $P$  and  $Q$ , find a minimal  $X$  such that  $P \star X \Rightarrow Q$ .

(In other words: what does  $P$  lack in order to ensure  $Q$ ?)

In general we cannot calculate a simple form for the solution  $X$ .  
But we can characterize it as follows:

$$X h = \forall h', h' \perp h \wedge P h' \Rightarrow Q(h' \uplus h)$$

This operation is written  $P \rightarrowstar Q$ , pronounced **magic wand**:

$$P \rightarrowstar Q \stackrel{\text{def}}{=} \lambda h. \forall h' \perp h, P h' \Rightarrow Q(h' \uplus h)$$

## Magic wand = separating implication

Separating implication  $\multimap$  is to separating conjunction  $\ast$  what plain implication  $\Rightarrow$  is to plain conjunction  $\wedge$ .

Adjunction property:

$$H \Rightarrow (P \multimap Q) \iff H \ast P \Rightarrow Q$$

Some other properties:

$P \ast (P \multimap Q)$	$\Rightarrow$	$Q$	(elimination)
$emp$	$\Rightarrow$	$P \multimap P$	(idempotence)
$(P \multimap Q) \ast (Q \multimap R)$	$\Rightarrow$	$P \multimap R$	(transitivity)
$(P \ast Q) \multimap R$	$=$	$P \multimap Q \multimap R$	(currying)
$(P \multimap Q) \ast R$	$\Rightarrow$	$P \multimap (Q \ast R)$	(distribution)

## The ramified consequence rule

$$\frac{\{P'\} c \{Q'\} \quad P \Rightarrow (P' \star (\forall v, Q' v \rightarrow \star Q v))}{\{P\} c \{Q\}}$$

Like the framed consequence rule, but with a canonical choice for the “frame”:  $R = \forall v, Q' v \rightarrow \star Q v$

Replaces the problem of finding  $R$  with the problem of reasoning with formulas that use  $\rightarrow \star$  and  $\star$ .

## Weakest preconditions

In separation logic, just like in Hoare logic, a command  $c$  with postcondition  $Q$  has a **weakest precondition**  $wp\ c\ Q$ , characterized by:

- It's a precondition:  $\{ wp\ c\ Q \} c \{ Q \}$
- It's the weakest: if  $\{ P \} c \{ Q \}$  then  $P \Rightarrow wp\ c\ Q$

We can define  $wp\ c\ Q$  in two ways:

from the operational semantics:  $wp\ c\ Q = \lambda h. \text{Term } c\ h\ Q$  (or Safe)

from the triples:  $wp\ c\ Q = \exists P. P \star \langle \{ P \} c \{ Q \} \rangle$



An equivalence with triples:

$$\{P\} c \{Q\} \quad \text{if and only if} \quad P \Rightarrow wp\ c\ Q$$

A view of deductive verification as a calculation, directed by the syntax of the command  $c$ :

“Given a command  $c$  and the specification  $Q$  of its results, what precondition should the initial state satisfy so that  $c$  executes without errors and the final state satisfies  $Q$ ?”

## A weakest precondition calculus

The rules of separation logic can be rephrased using *wp*:

$$Q \llbracket a \rrbracket \Rightarrow wp \ a \ Q$$

$$wp \ c \ (\lambda v. wp \ c' [x \leftarrow v] \ Q) \Rightarrow wp \ (\text{let } x = c \text{ in } c') \ Q$$

$$(\text{if } \llbracket b \rrbracket \text{ then } wp \ c_1 \ Q \text{ else } wp \ c_2 \ Q) \Rightarrow wp \ (\text{if } b \text{ then } c_1 \text{ else } c_2) \ Q$$

## A weakest precondition calculus

For the imperative constructs, the “small rules” lead to  $wp$  equations that are unusable, because they work only for postconditions  $Q$  of a very specific shape.

$$\text{emp} \Rightarrow wp(\text{alloc}(N)) (\lambda l. l \mapsto \_ \star \dots \star l + N - 1 \mapsto \_)$$

$$\llbracket a \rrbracket \mapsto x \Rightarrow wp(\text{get}(a)) (\lambda v. \langle v = x \rangle \star \llbracket a \rrbracket \mapsto x)$$

$$\llbracket a \rrbracket \mapsto \_ \Rightarrow wp(\text{set}(a, a')) (\lambda v. \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket)$$

$$\llbracket a \rrbracket \mapsto \_ \Rightarrow wp(\text{free}(a)) (\lambda v. \text{emp})$$

Time to wave the magic wand...

## Structural rules for weakest preconditions

The frame rule:

$$(wp\ c\ Q) \star R \Rightarrow wp\ c\ (\lambda v. Q\ v \star R)$$

The consequence rule:

$$\frac{\forall v, Q\ v \Rightarrow Q'\ v}{wp\ c\ Q \Rightarrow wp\ c\ Q'}$$

The ramified consequence rule:

$$wp\ c\ Q \star (\forall v, Q\ v \rightarrow Q'\ v) \Rightarrow wp\ c\ Q'$$

## A weakest precondition calculus

Applying ramification to the  $wp$  calculation for imperative constructs, we obtain  $wp$  equations usable for any postcondition  $Q$ .

$$\begin{aligned}\forall \ell, (\ell \mapsto \_ \star \dots \star \ell + N - 1 \mapsto \_) \multimap Q \ell &\Rightarrow wp(\text{alloc}(N)) Q \\ \exists x, \llbracket a \rrbracket \mapsto x \star (\llbracket a \rrbracket \mapsto x \multimap Q x) &\Rightarrow wp(\text{get}(a)) Q \\ \llbracket a \rrbracket \mapsto \_ \star (\forall v, \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \multimap Q v) &\Rightarrow wp(\text{set}(a, a')) Q \\ (\llbracket a \rrbracket \mapsto \_) \star (\forall v, Q v) &\Rightarrow wp(\text{free}(a)) Q\end{aligned}$$

## A taste of the “Iris proof mode”

In proof assistants such as Coq, we prefer to work with proof contexts

$$\frac{x_1 \dots x_n \quad H_1 \dots H_m}{P}$$

instead of formulas  $\forall x_1, \dots, x_n, H_1 \wedge \dots \wedge H_m \Rightarrow P$ .

## A taste of the “Iris proof mode”

In proof assistants such as Coq, we prefer to work with proof contexts

$$\frac{x_1 \dots x_n \quad H_1 \dots H_m}{P}$$

instead of formulas  $\forall x_1, \dots, x_n, H_1 \wedge \dots \wedge H_m \Rightarrow P$ .

A proof context in separation logic is:

$$\frac{\frac{x_1 \dots x_n \quad H_1 \dots H_m \quad (\text{standard hypotheses})}{P_1 \dots P_k} \quad (\text{spatial hypotheses})}{Q} \quad (\text{goal})$$

It stands for  $\forall x_i, H_1 \wedge \dots \wedge H_m \Rightarrow P_1 \star \dots \star P_k \Rightarrow Q$ .

## Some introduction rules

$$\frac{\frac{\dots}{\dots}}{P_1 * \dots * P_n \rightarrow Q} \rightsquigarrow \frac{\dots}{\dots \frac{P_1 \dots P_n}{Q}}$$

$$\frac{\frac{\dots}{\dots \langle H \rangle}}{\dots} \rightsquigarrow \frac{\dots H}{\dots}$$

$$\frac{\dots}{\dots \exists x, P x} \rightsquigarrow \frac{\dots x}{\dots P x}$$



## Weakest preconditions $\approx$ symbolic execution

$wp\ c\ Q \approx$  “we do  $c$ , then we will have  $Q$ ”.

The postcondition  $Q$  plays the role of a continuation, memorizing what comes next during execution.

$$\begin{array}{ccc} \frac{\dots}{\dots} & \rightsquigarrow & \frac{\dots}{\dots} \\ \hline \frac{\dots}{wp\ (\text{let } x = c_1 \text{ in } c_2)\ Q} & & \frac{\dots}{wp\ c_1\ (\lambda v. wp\ c_2[x \leftarrow v])\ Q} \\ \\ \frac{\dots}{\dots} & \rightsquigarrow \dots \rightsquigarrow & \frac{\dots}{\dots} \\ \hline \frac{\dots}{wp\ a\ (\lambda v. wp\ c_2[x \leftarrow v])\ Q} & \rightsquigarrow & \frac{\dots}{wp\ c_2[x \leftarrow a]\ Q} \end{array}$$

## Symbolic execution of memory operations

The *wp* rules for the memory operations become clearer:

$$\exists x, \llbracket a \rrbracket \mapsto x \star (\llbracket a \rrbracket \mapsto x \rightarrow \star Q x) \Rightarrow wp(\text{get}(a)) Q$$

$$\frac{\dots}{\llbracket a \rrbracket \mapsto x \quad \dots} \quad \rightsquigarrow \quad \frac{\dots}{\llbracket a \rrbracket \mapsto x \quad \dots}$$
$$\frac{}{wp(\text{get}(a)) Q} \quad \quad \quad \frac{}{Q x}$$

$$\llbracket a \rrbracket \mapsto \_ \star (\forall v, \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \rightarrow \star Q v) \Rightarrow wp(\text{set}(a, a')) Q$$

$$\frac{\dots}{\llbracket a \rrbracket \mapsto \_ \quad \dots} \quad \rightsquigarrow \quad \frac{\dots \quad v}{\llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \quad \dots}$$
$$\frac{}{wp(\text{set}(a, a')) Q} \quad \quad \quad \frac{}{Q v}$$

# Symbolic execution of memory operations

$\forall l, (l \mapsto \_ * \dots * l + N - 1 \mapsto \_) \multimap Q \Rightarrow wp(\text{alloc}(N)) Q$

$$\frac{\dots}{\dots} \rightsquigarrow \frac{\dots \quad l \quad \dots \quad l + N - 1}{\dots \quad l \mapsto \_ \quad \dots \quad l + N - 1 \mapsto \_}$$

$$\frac{}{wp(\text{alloc}(N)) Q} \qquad \frac{}{Q l}$$

$(\llbracket a \rrbracket \mapsto \_) \multimap (\forall v, Q v) \Rightarrow wp(\text{free}(a)) Q$

$$\frac{\dots}{\llbracket a \rrbracket \mapsto \_ \quad \dots} \rightsquigarrow \frac{\dots \quad v}{\dots}$$

$$\frac{}{wp(\text{free}(a)) Q} \qquad \frac{}{Q v}$$

## **Partial permissions**

---

## Read-only sharing

Several processes access a shared data structure without synchronization, but do not modify the data structure.

$$x := \dots T[i] \dots \parallel y := \dots T[i] \dots \parallel z := \dots T[i] \dots$$

This is safe:

- No race conditions (two simultaneous reads from the same location produce a well-defined result).
- Cf. the Rust motto: “shared xor mutable”.

## Read-only sharing

Several processes access a shared data structure without synchronization, but do not modify the data structure.

$$x := \dots T[i] \dots \parallel y := \dots T[i] \dots \parallel z := \dots T[i] \dots$$

This is efficient:

- No need to copy the data structure in each process.  
( $\neq$  distributed-memory parallelism).

## Read-only sharing

Several processes access a shared data structure without synchronization, but do not modify the data structure.

$$x := \dots T[i] \dots \parallel y := \dots T[i] \dots \parallel z := \dots T[i] \dots$$

This cannot be expressed in basic separation logic!

- Outside a critical section or atomic section, a memory location is accessible (for reads and for writes) by only one process.

$$\frac{\{P_1\} c_1 \{ \lambda \dots Q_1 \} \quad \{P_2\} c_2 \{ \lambda \dots Q_2 \}}{\{P_1 \star P_2\} c_1 \parallel c_2 \{ \lambda \dots Q_1 \star Q_2 \}}$$

- Cf. the “no race conditions” theorem

## A permission model

The assertion  $\ell \mapsto v$ , “location  $\ell$  contains value  $v$ ”, can also be read as a permission to access location  $\ell$  for reading, writing, or freeing.

Naive idea: distinguish two permissions

Full permission:  $\ell \xrightarrow{1} v$  (read, write, free)

Read-only permission:  $\ell \xrightarrow{R} v$  (read)



## A permission model

The “small rule” for `get` accepts both permissions. The other small rules produce or require full permissions.

$$\begin{array}{lll} \{ \text{emp} \} & \text{alloc}(N) & \{ \lambda l. l \xrightarrow{1} \_ \star \dots \star l + N - 1 \xrightarrow{1} \_ \} \\ \{ \llbracket a \rrbracket \xrightarrow{\pi} x \} & \text{get}(a) & \{ \lambda v. \langle v = x \rangle \star \llbracket a \rrbracket \xrightarrow{\pi} x \} \quad (\pi \in \{1, R\}) \\ \{ \llbracket a \rrbracket \xrightarrow{1} \_ \} & \text{set}(a, a') & \{ \lambda v. \llbracket a \rrbracket \xrightarrow{1} \llbracket a' \rrbracket \} \\ \{ \llbracket a \rrbracket \xrightarrow{1} \_ \} & \text{free}(a) & \{ \lambda v. \text{emp} \} \end{array}$$

A full permission can be **weakened**:

$$l \xrightarrow{1} v \Rightarrow l \xrightarrow{R} v$$

A read-only permission can be **duplicated**:

$$l \xrightarrow{R} v = l \xrightarrow{R} v \star l \xrightarrow{R} v$$

## Example of read-only sharing

```
let t = alloc(1) in
```

```
  set(t, f(x));
```

$$\{t \xrightarrow{1} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x)\}$$

$\{t \xrightarrow{R} f(x)\}$		$\{t \xrightarrow{R} f(x)\}$
...get(t)...		...get(t)...
...		...
$\{t \xrightarrow{R} f(x) * Q_1\}$		$\{t \xrightarrow{R} f(x) * Q_2\}$

$$\{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x) * Q_1 * Q_2\}$$

## Example of read-only sharing

```
let t = alloc(1) in
```

```
  set(t, f(x));
```

$$\{t \xrightarrow{1} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x)\}$$

$\{t \xrightarrow{R} f(x)\}$		$\{t \xrightarrow{R} f(x)\}$
...get(t)...		...get(t)...
...		...
$\{t \xrightarrow{R} f(x) * Q_1\}$		$\{t \xrightarrow{R} f(x) * Q_2\}$

$$\{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x) * Q_1 * Q_2\}$$

Problem: we cannot free  $t$  at the end of this code!

## Fractional permissions

$$\frac{1}{2} + \frac{1}{2} = 1$$

Boyland (2003): permissions  $\pi$  are rational numbers in  $(0, 1]$ .

- $\pi = 1$ : full permission.
- $0 < \pi < 1$ : read-only permission.

The law for splitting and recombining permissions:

$$l \xrightarrow{\pi+\pi'} v = l \xrightarrow{\pi} v \star l \xrightarrow{\pi'} v \quad \text{if } \pi + \pi' \in (0, 1]$$

## Read-only sharing using fractional permissions

```
let t = alloc(1) in  
set(t, f(x));
```

$$\{t \mapsto^1 f(x)\} \Rightarrow \{t \mapsto^{1/2} f(x) * t \mapsto^{1/2} f(x)\}$$

$$\begin{array}{c} \{t \mapsto^{1/2} f(x)\} \\ \dots \text{get}(t) \dots \\ \dots \\ \{t \mapsto^{1/2} f(x) * Q_1\} \end{array} \quad \parallel \quad \begin{array}{c} \{t \mapsto^{1/2} f(x)\} \\ \dots \text{get}(t) \dots \\ \dots \\ \{t \mapsto^{1/2} f(x) * Q_2\} \end{array}$$

$$\{t \mapsto^{1/2} f(x) * t \mapsto^{1/2} f(x) * Q_1 * Q_2\} \Rightarrow \{t \mapsto^1 f(x) * Q_1 * Q_2\}$$

`free(t)`

$$\{Q_1 * Q_2\}$$

A set  $\Pi$  equipped with a **partial** operation  $\oplus$  to combine two permissions. The operation is

**commutative**

$$\pi_1 \oplus \pi_2 = \pi_2 \oplus \pi_1$$

and **associative**

$$(\pi_1 \oplus \pi_2) \oplus \pi_3 = \pi_1 \oplus (\pi_2 \oplus \pi_3).$$

Note: read  $\pi_1 \oplus \pi_2 = \pi$  as “the combination  $\pi_1 \oplus \pi_2$  is defined and equal to  $\pi$ ”.

## Heaps with permissions

A heap  $h$  is a finite function from locations to pairs  $(\pi, v)$  of a permission and a value.

We define the combination of two such pairs as:

$$(\pi_1, v_1) \oplus (\pi_2, v_2) = (\pi_1 \oplus \pi_2, v_1) \text{ if } v_1 = v_2 \text{ and } \pi_1 \oplus \pi_2 \text{ is defined}$$

## Heaps with permissions

The combination  $h_1 \oplus h_2$  of two heaps is defined if

$$h_1 \perp h_2 \stackrel{\text{def}}{=} \forall \ell \in \text{Dom}(h_1) \cap \text{Dom}(h_2), h_1(\ell) \oplus h_2(\ell) \text{ is defined}$$

The combination is defined by

$$\text{Dom}(h_1 \oplus h_2) = \text{Dom}(h_1) \cup \text{Dom}(h_2)$$
$$(h_1 \oplus h_2)(\ell) = \begin{cases} h_1(\ell) \oplus h_2(\ell) & \text{if } \ell \in \text{Dom}(h_1) \cap \text{Dom}(h_2) \\ h_1(\ell) & \text{if } \ell \in \text{Dom}(h_1) \setminus \text{Dom}(h_2) \\ h_2(\ell) & \text{if } \ell \in \text{Dom}(h_2) \setminus \text{Dom}(h_1) \end{cases}$$

Generalizes the notion of disjoint union  $h_1 \uplus h_2$ .



## Separating conjunction, separating implication

The usual definitions, using heap combination  $\oplus$  instead of disjoint union  $\uplus$ .

$$P \star Q = \lambda h. \exists h_1, h_2, h = h_1 \oplus h_2 \wedge P h_1 \wedge Q h_2$$

$$P \multimap Q = \lambda h. \exists h_1, h_2, h_2 = h \oplus h_1 \wedge P h_1 \wedge Q h_2$$

In particular,  $\ell \xrightarrow{\pi_1} v_1 \star \ell \xrightarrow{\pi_2} v_2$  holds if and only if

$\pi_1 \oplus \pi_2$  is defined,  $v_1 = v_2$ , and  $\ell \xrightarrow{\pi_1 \oplus \pi_2} v_1$ .

An interesting property: every command  $c$  provable with a precondition “read-only permission on location  $l$ ” cannot modify  $l$ .

Proof sketch in the sequential case:

By way of contradiction, assume  $\{l \stackrel{1/2}{\mapsto} v\} c \{ \lambda_. l \stackrel{1/2}{\mapsto} v' \}$  with  $v' \neq v$ .

By framing with  $l \stackrel{1/2}{\mapsto} v$ , we get

$$\{l \stackrel{1}{\mapsto} v\} c \{ \lambda_. l \stackrel{1/2}{\mapsto} v' \star l \stackrel{1/2}{\mapsto} v \}$$

The precondition is true, but the postcondition is always false, since location  $l$  cannot contain both  $v$  and  $v'$ .

## Passivity and atomic sections

In the presence of atomic sections or critical sections, the passivity property is less clear-cut.

Indeed, if the shared-memory invariant gives us the missing permission  $\ell \stackrel{1/2}{\mapsto} \_$ , we can derive

$$\ell \stackrel{1/2}{\mapsto} \_ \vdash \{ \ell \stackrel{1/2}{\mapsto} v \} \text{atomic}(\text{set}(\ell, v')) \{ \lambda \_. \ell \stackrel{1/2}{\mapsto} v' \}$$

even if  $v' \neq v$ .

## Read-only sharing with counting permissions

Another permission schema, better suited to programs using readers-writer locks.

Permissions  $\pi$  are integers  $\geq -1$ :

- 0: full permission (get, set, alloc, free)
- -1: read-only permission (get)
- $n > 0$ : number of read-only permissions that were granted.

We have:

$$\ell \xrightarrow{n} v = \ell \xrightarrow{n+1} v \star \ell \xrightarrow{-1} v \quad \text{if } n \geq 0$$

	Readers	Writer
{ emp }	$P(\text{read});$ $\text{count} := \text{count} + 1;$ if $\text{count} = 1$ then $P(\text{write});$ $V(\text{read});$	
{ $b \overset{-1}{\mapsto} \_$ }	read $b$	{ emp }
{ $b \overset{-1}{\mapsto} \_$ }	$P(\text{read});$ $\text{count} := \text{count} - 1;$ if $\text{count} = 0$ then $V(\text{write});$ $V(\text{read});$	{ $b \overset{0}{\mapsto} \_$ }
{ emp }		{ emp }

Invariant for write:  $b \overset{0}{\mapsto} \_$

Invariant for read:  $\exists n, \text{count} \overset{0}{\mapsto} n \star (\langle n = 0 \rangle \vee \langle n > 0 \rangle \star b \overset{n}{\mapsto} \_)$

## Ghost code

---

Two ways to facilitate writing specifications as Hoare triples.

**Auxiliary variables:** mathematical variables  $\alpha, \beta, \dots$  universally quantified before the triple.

$$\forall \alpha, \beta, \{ x = \alpha \wedge y = \beta \} \text{ if } x < y \text{ then } x := y \{ x = \max(\alpha, \beta) \}$$

**Ghost variables:** variables from the programming language that do not appear in the program.

$$\{ z = x \} \text{ if } x < y \text{ then } x := y \{ x = \max(z, y) \}$$



To make verification easier, we can add **ghost code**: commands that modify ghost variables but have no effects on program variables.

This ghost code can be removed before execution, since normal (non-ghost) code does not depend on ghost variables.



## Example: remainder of Euclidean division

```

                                {  $a \geq 0$  }
r := a;

while r  $\geq$  b do
                                {  $r \geq 0 \wedge \exists q, a = b \cdot q + r$  }

    r := r - b
done
                                {  $r = a \bmod b$  }
```

Automated theorem provers sometimes have a hard time with existential quantification...

## Example: remainder of Euclidean division

```

                                     { a ≥ 0 }
r := a;
👻 q := 0;
while r ≥ b do
                                     { r ≥ 0 ∧ a = b · q + r }
    👻 q := q + 1;
    r := r - b
done
                                     { r = a mod b }
```

The ghost code computes the appropriate value for  $q$ .  
The theorem prover only has to check this value.

## Example: a recursive graph traversal

Like in lecture #3: mark all nodes reachable from the root  $r$ .

```
def DFS r =  
  if MARK[r] = 0 then begin  
    MARK[r] := 1;  
    for i = 0 to ARITY[r] - 1 do DFS(CHILD[r][i]) done  
  end
```

It is surprisingly hard to prove this code correct!

## Example: a recursive graph traversal

We reintroduce the worklist  $W$  as a ghost variable.  
( $W \approx$  the nodes that remain to be traversed)

```
def DFSREC  $p$  =  
  👻  $W := W \setminus \{p\}$ ;  
  if MARK[ $p$ ] = 0 then begin  
    MARK[ $p$ ] := 1;  
    👻  $W := W \cup \{\text{CHILD}[p][i] \mid 0 \leq i < \text{ARITY}[p]\}$ ;  
    for  $i = 0$  to ARITY[ $p$ ] - 1 do DFS (CHILD[ $p$ ][ $i$ ]) done  
  end  
def DFS  $r$  =  
  👻  $W := \{r\}$ ;  
  DFSREC  $r$ 
```

## Example: a recursive graph traversal

We can then show the invariant

$$\forall x, \text{path}(r, x) \iff \text{MARK}[x] = 1 \vee \exists p \in W, \text{path}(p, x)$$

and conclude

$$\{ \forall x, \text{MARK}[x] = 0 \} \text{ DFS } r \{ \forall x, \text{path}(r, x) \iff \text{MARK}[x] = 1 \}$$

Note: ghost code is not always executable, and ghost variables can be of a type that is not expressible in the programming language! Here, we used mathematical sets for  $W$ .

## Ghost code and concurrency

In a concurrent program, we can use ghost code and ghost variables to keep track of the actions of each process.

Example: producer/consumer.

```

                                👻  $PR := \epsilon; CO := \epsilon;$ 
while true do                  || while true do
  compute x;                   ||   let  $y = consume()$  in
                                ||   👻  $CO := CO \cdot y$ 
                                ||   use  $y$ 
  👻  $PR := PR \cdot x$            || done
  produce( $x$ );
done                            ||
```

The ghost lists  $PR$  and  $CO$  keep track of the data produced or consumed.

## The puzzle: $1 + 1 = 2$ ?

```
set(n, 0);  
atomic(incr(n)) || atomic(incr(n))
```

With  $incr(p) \stackrel{def}{=} \text{let } x = \text{get}(p) \text{ in set}(p, x + 1)$ .

In the previous lecture, we saw how to prove the safety of this code and the fact that  $n \geq 0$  at the end, using the resource invariant  $J = \exists x, n \mapsto x \star \langle x \geq 0 \rangle$ .

But how can we prove full correctness? that is,  $n = 2$  at the end?

## Tracing processes with ghost code

```
set(n, 0);
```




```
 set(a, 0); set(b, 0);
```

```
atomic(incr(n);  incr(a)) || atomic(incr(n);  incr(b))
```

*a* represents the contribution of the left process to the sum *n*,  
*b* represents that of the right process.



## Tracing processes with ghost code

```
set(n, 0);  
     set(a, 0); set(b, 0);  
atomic(incr(n);  incr(a)) || atomic(incr(n);  incr(b))
```

$a$  represents the contribution of the left process to the sum  $n$ ,  
 $b$  represents that of the right process.

We would like to reflect this in the invariant:

$$\exists x, y, a \mapsto x \star b \mapsto y \star n \mapsto x + y.$$

But this requires  $a$  and  $b$  to belong to the shared state.

We would like to show  $a \mapsto 1$  and  $b \mapsto 1$  at the end.

But this requires that  $a$  belongs to the left process and  $b$  to the right process.

## Fractional permissions to the rescue!

Consider the resource invariant

$$J = \exists x, y, a \stackrel{1/2}{\mapsto} x \star b \stackrel{1/2}{\mapsto} y \star n \stackrel{1}{\mapsto} x + y$$

We have:

$$\{J \star a \stackrel{1/2}{\mapsto} x\} \Rightarrow \\ \{a \stackrel{1}{\mapsto} x \star \exists y, b \stackrel{1/2}{\mapsto} y \star n \stackrel{1}{\mapsto} x + y\}$$

*incr*(n);

*incr*(a);

$$\{a \stackrel{1}{\mapsto} x + 1 \star \exists y, b \stackrel{1/2}{\mapsto} y \star n \stackrel{1}{\mapsto} x + 1 + y\} \\ \Rightarrow \{J \star a \stackrel{1/2}{\mapsto} x + 1\}$$

Therefore,  $J \vdash \{a \stackrel{1/2}{\mapsto} x\} \text{atomic}(\text{incr}(n); \text{incr}(a)) \{a \stackrel{1/2}{\mapsto} x + 1\}$

## Fractional permissions to the rescue

We can then derive:

$$\begin{array}{c} \text{set}(n, 0); \text{set}(a, 0); \text{set}(b, 0); \\ \{n \overset{1}{\mapsto} 0 * a \overset{1}{\mapsto} 0 * b \overset{1}{\mapsto} 0\} \Rightarrow \{J * a \overset{1/2}{\mapsto} 0 * b \overset{1/2}{\mapsto} 0\} \\ \{a \overset{1/2}{\mapsto} 0\} \quad \parallel \quad \{b \overset{1/2}{\mapsto} 0\} \\ \text{atomic}(\text{incr}(n); \text{incr}(a)) \quad \parallel \quad \text{atomic}(\text{incr}(n); \text{incr}(b)) \\ \{a \overset{1/2}{\mapsto} 1\} \quad \parallel \quad \{b \overset{1/2}{\mapsto} 1\} \\ \{J * a \overset{1/2}{\mapsto} 1 * b \overset{1/2}{\mapsto} 1\} \Rightarrow \{n \overset{1}{\mapsto} 2 * a \overset{1}{\mapsto} 1 * b \overset{1}{\mapsto} 1\} \end{array}$$

Therefore,  $n = 2$  in the end!

## Fractional permissions to the rescue

We can then derive:

$$\begin{array}{c} \text{set}(n, 0); \text{set}(a, 0); \text{set}(b, 0); \\ \{n \xrightarrow{1} 0 * a \xrightarrow{1} 0 * b \xrightarrow{1} 0\} \Rightarrow \{J * a \xrightarrow{1/2} 0 * b \xrightarrow{1/2} 0\} \\ \\ \left\{ \begin{array}{l} \{a \xrightarrow{1/2} 0\} \\ \text{atomic}(\text{incr}(n); \text{incr}(a)) \\ \{a \xrightarrow{1/2} 1\} \end{array} \right\} \parallel \left\{ \begin{array}{l} \{b \xrightarrow{1/2} 0\} \\ \text{atomic}(\text{incr}(n); \text{incr}(b)) \\ \{b \xrightarrow{1/2} 1\} \end{array} \right\} \\ \{J * a \xrightarrow{1/2} 1 * b \xrightarrow{1/2} 1\} \Rightarrow \{n \xrightarrow{1} 2 * a \xrightarrow{1} 1 * b \xrightarrow{1} 1\} \end{array}$$

Therefore,  $n = 2$  in the end!

This is an elementary example of a very general technique:  
protocols governing the evolutions of ghost states

## **Storable locks**

---

# Fine-grained parallelism

Two kinds of mutual exclusion:

- **Coarse-grained:** a (global) lock protects the whole data structure.

Described well by the resource model of O'Hearn's concurrent separation logic.

- **Fine-grained:** one lock per memory block comprised in the data structure.

Need to reason about locks that are stored in memory, inside the block that they protect against simultaneous accesses.

## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes “2”, the other removes “5”.



## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes "2", the other removes "5".



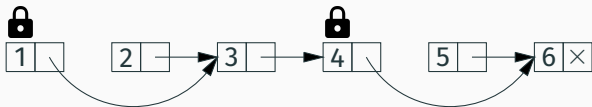


## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes "2", the other removes "5".

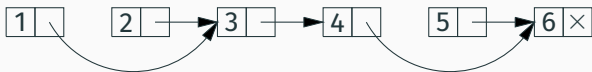


## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes "2", the other removes "5".



## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes “2”, the other removes “5”.



Locking one node at a time is not enough!

Example: one process removes “3”, the other removes “4”.

## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes “2”, the other removes “5”.



Locking one node at a time is not enough!

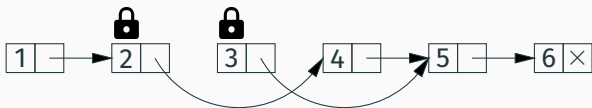
Example: one process removes “3”, the other removes “4”.

## Example: singly-linked lists

```
struct cell { lock lck; int val; struct cell * next; };
```

By locking nodes one after the other, we can operate over the list in parallel.

Example: one process removes “2”, the other removes “5”.



Locking one node at a time is not enough!

Example: one process removes “3”, the other removes “4”.

The result can be [1; 2; 4; 5; 6] instead of [1; 2; 5; 6].

# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.



# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.



# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.





# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.



# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.



# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.



# Hand-over-hand locking

To modify a node, we must have locked the node **as well as the node before**.

Example: removal of “4”.



# Specification of stored locks

Two new assertions:

$l \bullet \xrightarrow{\pi} RI$  “at location  $l$ , with permission  $\pi$ , there is a lock that protects the resource described by  $RI$ ”

$\mathfrak{L} l$  “the lock at location  $l$  is locked by the current process”

The “small rules” for locks:

$$\begin{array}{l} \{ l \bullet \xrightarrow{\pi} RI \} \quad \text{lock}(l) \quad \{ l \bullet \xrightarrow{\pi} RI \star \mathfrak{L} l \star RI \} \\ \{ l \bullet \xrightarrow{\pi} RI \star \mathfrak{L} l \star RI \} \quad \text{unlock}(l) \quad \{ l \bullet \xrightarrow{\pi} RI \} \\ \{ l \xrightarrow{1} \_ \star RI \} \quad \text{initlock}(l) \quad \{ l \bullet \xrightarrow{1} RI \} \\ \{ l \bullet \xrightarrow{1} RI \} \quad \text{destroylock}(l) \quad \{ l \xrightarrow{1} \_ \star RI \} \end{array}$$

## Representation predicate for sorted lists

(Following Gotsman, Berdine, Cook, Rinetzky and Sagiv, 2007.

See Jacobs and Piessens, 2011, for a more fine-grained specification.)

We add a sentinel  $-\infty$  at the beginning and another  $+\infty$  at end.

$$\begin{aligned} list(p, n) = & (\langle n = +\infty \rangle \star p.val \xrightarrow{1} n \star p.next \xrightarrow{1} \text{NULL}) \\ & \vee (\exists q, n', \langle n < n' \rangle \star p.val \xrightarrow{1} n \star p.next \xrightarrow{1} q \\ & \star p.lock \bullet \xrightarrow{1} list(q, n')) \end{aligned}$$

$$\begin{aligned} listhead(p, \pi) = & \exists q, n, p.val \xrightarrow{\pi} -\infty \star p.next \xrightarrow{\pi} q \\ & \star p.lock \bullet \xrightarrow{\pi} list(q, n) \end{aligned}$$

The head of the list (the  $-\infty$  sentinel) is shared ( $\pi < 1$ ). The other list nodes are in exclusive access mode, protected by the lock contained in the previous node.

## Summary

---

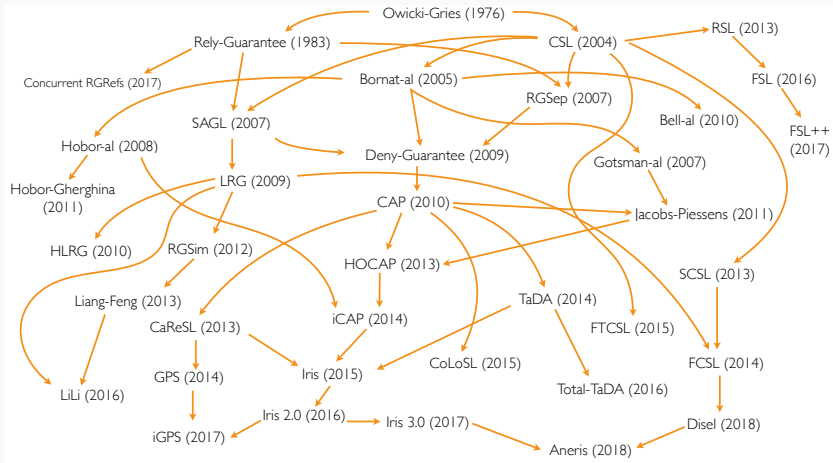
We have seen a few extensions to separation logic, both sequential and concurrent.

Many other extensions have been studied in the last 20 years:

- “first-class  $X$ ” for various values of  $X$ : functions, Hoare triples, process ID, ...
- Modular reasoning: for instance, interactions between an arbitrary number of processes, not just 2.
- Verification of advanced concurrent algorithms: optimistic locking, lock-free algorithms, etc.



# A proliferation of program logics



(Diagram by Ilya Sergey)

## Assertions that talk about many different things

- Purely logical facts  $\langle P \rangle$
- Facts about variables  $x = \alpha$  (in Hoare logic)
- Facts about the memory heap  $\text{emp}, \ell \mapsto v, \ell \mapsto \_$
- The same plus permissions  $\ell \overset{\pi}{\mapsto} v$
- Facts about locks  $\ell \bullet \overset{\pi}{\rightarrow} RI, \text{🔒} \ell$
- Facts about ghost states.
- Time credits
- Transition systems
- What else?

## Iris: a consolidation around four notions

1– **Resource algebras**, previously called *partial commutative monoids*.

(An operation  $\oplus$  commutative, associative, **partial**, representing the combination of two compatible “things”.)

2– **Ghost transitions**, generalizing ghost code.

3– **Invariants**, generalizing the various kinds of resource invariants previously mentioned.

4– A systematic use of **step indexing** and the “**later**” modality ( $\triangleright$ ) to work around circular definitions of higher-order notions.

## References

---

### All about Iris:

- Papers, tutorials, Coq development:  
<https://iris-project.org/>

### Partial permissions:

- R. Bornat, C. Calcagno, P. O'Hearn, M. Parkinson, *Permission accounting in separation logic*, POPL 2005.

### Storable locks:

- A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv, *Local reasoning for storable locks and threads*, APLAS 2007.