Program logics, fifth lecture

Some extensions of separation logic

Xavier Leroy

2021-04-01

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" $\{\, \ell \mapsto \_ \,\}\ \mathtt{set}(\ell, v)\ \{\, \lambda\_.\ \ell \mapsto v \,\}$
- or a "small specification" for a function
  $\{\, \mathit{list}(w, p) \,\}\ \mathtt{reverse}(p)\ \{\, \lambda r.\ \mathit{list}(\mathit{rev}(w), r) \,\}$

in a bigger context, such as

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

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 $\{\, P \,\}\ c\ \{\, Q \,\}$.

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

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:

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

This operation is written $P \mathrel{-\!\!\star} Q$, pronounced magic wand:

$$P \mathrel{-\!\!\star} Q \ \stackrel{def}{=}\ \lambda h.\ \forall h' \perp h, P\ h' \Rightarrow Q(h' \uplus h)$$

Magic wand = separating implication

Separating implication $-\!\!\ast$ is to separating conjunction $\star$ what plain implication $\Rightarrow$ is to plain conjunction $\wedge$.

Adjunction property:

$$H \Rightarrow (P -\!\!\ast\, Q) \quad\Longleftrightarrow\quad H \star P \Rightarrow Q$$

Some other properties:

$$\begin{array}{rcll}
P \star (P -\!\!\ast\, Q) & \Rightarrow & Q & \text{(elimination)} \\
emp & \Rightarrow & P -\!\!\ast\, P & \text{(idempotence)} \\
(P -\!\!\ast\, Q) \star (Q -\!\!\ast\, R) & \Rightarrow & P -\!\!\ast\, R & \text{(transitivity)} \\
(P \star Q) -\!\!\ast\, R & = & P -\!\!\ast\, Q -\!\!\ast\, R & \text{(currying)} \\
(P -\!\!\ast\, Q) \star R & \Rightarrow & P -\!\!\ast\, (Q \star R) & \text{(distribution)}
\end{array}$$

The ramified consequence rule

$$\frac{\{\,P'\,\}\ c\ \{\,Q'\,\} \qquad P \Rightarrow (P' \ast (\forall v, Q'\ v \mathrel{-\!\!\ast} Q\ v))}{\{\,P\,\}\ c\ \{\,Q\,\}}$$

Like the framed consequence rule, but with a canonical choice for the "frame": $R = \forall v, Q'\ v \mathrel{-\!\!\ast} Q\ v$

Replaces the problem of finding R with the problem of reasoning with formulas that use $\mathrel{-\!\!\ast}$ and $\ast$.

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: $wp\ c\ Q \;=\; \lambda h.\ \mathtt{Term}\ c\ h\ Q$ (or $\mathtt{Safe}$)

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

Weakest preconditions

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*:

$$
\begin{aligned}
Q\,[\![a]\!] &\;\Rightarrow\; wp\ a\ Q \\
wp\ c\ (\lambda v.\ wp\ c'[x \leftarrow v]\ Q) &\;\Rightarrow\; wp\ (\mathtt{let}\ x = c\ \mathtt{in}\ c')\ Q \\
(\text{if } [\![b]\!] \text{ then } wp\ c_1\ Q \text{ else } wp\ c_2\ Q) &\;\Rightarrow\; wp\ (\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)\ Q
\end{aligned}
$$

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.

$$\begin{array}{rcl}
\texttt{emp} & \Rightarrow & wp\,(\texttt{alloc}(N))\,(\lambda\ell.\,\ell \mapsto _ \star \cdots \star \ell + N - 1 \mapsto _) \\
[\![a]\!] \mapsto x & \Rightarrow & wp\,(\texttt{get}(a))\,(\lambda v.\,\langle v = x\rangle \star [\![a]\!] \mapsto x) \\
[\![a]\!] \mapsto _ & \Rightarrow & wp\,(\texttt{set}(a, a'))\,(\lambda v.\,[\![a]\!] \mapsto [\![a']\!]) \\
[\![a]\!] \mapsto _ & \Rightarrow & wp\,(\texttt{free}(a))\,(\lambda v.\,\texttt{emp})
\end{array}$$

Time to wave the magic wand…

Structural rules for weakest preconditions

The frame rule:

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

The consequence rule:

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

The ramified consequence rule:

$$wp\ c\ Q \ast (\forall v,\ Q\ v \mathrel{-\!\!\ast} Q'\ v) \Rightarrow wp\ c\ Q'$$

A weakest precondition calculus

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

$$
\begin{array}{rcl}
\forall \ell,\ (\ell \mapsto \_ \star \cdots \star \ell + N - 1 \mapsto \_) \mathrel{-\!\!\star} Q\ \ell & \Rightarrow & wp\ (\texttt{alloc}(N))\ Q \\
\exists x,\ [\![a]\!] \mapsto x \star ([\![a]\!] \mapsto x \mathrel{-\!\!\star} Q\ x) & \Rightarrow & wp\ (\texttt{get}(a))\ Q \\
[\![a]\!] \mapsto \_ \star (\forall v,\ [\![a]\!] \mapsto [\![a']\!] \mathrel{-\!\!\star} Q\ v) & \Rightarrow & wp\ (\texttt{set}(a, a'))\ Q \\
([\![a]\!] \mapsto \_) \star (\forall v, Q\ v) & \Rightarrow & wp\ (\texttt{free}(a))\ Q
\end{array}
$$

A taste of the "Iris proof mode"

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

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

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

A taste of the "Iris proof mode"

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

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

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:

$$\begin{array}{cl} x_1 \dots x_n \;\; H_1 \dots H_m & \text{(standard hypotheses)} \\ \hline P_1 \dots P_k & \text{(spatial hypotheses)} \\ \hline Q & \text{(goal)} \end{array}$$

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

Some introduction rules

$$\dfrac{\dfrac{\cdots}{\cdots}}{P_1 \ast \cdots \ast P_n \mathrel{-\!\!\ast} Q} \quad \rightsquigarrow \quad \dfrac{\dfrac{\cdots}{\cdots \quad P_1 \ldots P_n}}{Q}$$

$$\dfrac{\dfrac{\cdots}{\cdots \quad \langle H \rangle}}{\cdots} \quad \rightsquigarrow \quad \dfrac{\dfrac{\cdots \quad H}{\cdots}}{\cdots}$$

$$\dfrac{\dfrac{\cdots}{\cdots \quad \exists x,\ P\,x}}{\cdots} \quad \rightsquigarrow \quad \dfrac{\dfrac{\cdots \quad x}{\cdots \quad P\,x}}{\cdots}$$

Weakest preconditions $\approx$ symbolic execution

$\texttt{wp}\ c\ Q \approx$ "we do c, then we will have Q".

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

$$\cfrac{\cfrac{\dots}{\dots}}{wp\ (\texttt{let}\ x = c_1\ \texttt{in}\ c_2)\ Q} \quad \rightsquigarrow \quad \cfrac{\cfrac{\dots}{\dots}}{wp\ c_1\ (\lambda v.\ wp\ c_2[x \leftarrow v]\ Q)}$$

$$\rightsquigarrow \dots \rightsquigarrow \quad \cfrac{\cfrac{\dots}{\dots}}{wp\ a\ (\lambda v.\ wp\ c_2[x \leftarrow v]\ Q)} \quad \rightsquigarrow \quad \cfrac{\cfrac{\dots}{\dots}}{wp\ c_2[x \leftarrow a]\ Q}$$

Symbolic execution of memory operations

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

$$\exists x,\ [\![a]\!] \mapsto x \star ([\![a]\!] \mapsto x \mathrel{-\!\!\star} Q\,x) \Rightarrow wp\ (\mathtt{get}(a))\ Q$$

$$\dfrac{\dfrac{\ldots}{[\![a]\!] \mapsto x \quad \ldots}}{wp\ (\mathtt{get}(a))\ Q} \quad\rightsquigarrow\quad \dfrac{\dfrac{\ldots}{[\![a]\!] \mapsto x \quad \ldots}}{Q\,x}$$

$$[\![a]\!] \mapsto \_ \star (\forall v,\ [\![a]\!] \mapsto [\![a']\!] \mathrel{-\!\!\star} Q\,v) \Rightarrow wp\ (\mathtt{set}(a, a'))\ Q$$

$$\dfrac{\dfrac{\ldots}{[\![a]\!] \mapsto \_ \quad \ldots}}{wp\ (\mathtt{set}(a, a'))\ Q} \quad\rightsquigarrow\quad \dfrac{\dfrac{\ldots \quad v}{[\![a]\!] \mapsto [\![a']\!] \quad \ldots}}{Q\,v}$$

Symbolic execution of memory operations

$$\forall \ell,\ (\ell \mapsto \_ \ast \cdots \ast \ell + N - 1 \mapsto \_) \mathrel{-\!\!\ast} Q\ \ell \Rightarrow wp\ (\texttt{alloc}(N))\ Q$$

$$\dfrac{\dfrac{\dots}{\dots}}{wp\ (\texttt{alloc}(N))\ Q} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dots \quad \ell}{\dots \quad \ell \mapsto \_ \quad \dots \quad \ell + N - 1 \mapsto \_}}{Q\ \ell}$$

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

$$\dfrac{\dfrac{\dots}{\llbracket a \rrbracket \mapsto \_ \quad \dots}}{wp\ (\texttt{free}(a))\ Q} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dots \quad v}{\dots}}{Q\ v}$$

Partial permissions

Read-only sharing

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

$$x := \ldots T[i] \ldots \;\Big\|\; y := \ldots T[i] \ldots \;\Big\|\; z := \ldots T[i] \ldots$$

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.

$$x := \ldots T[i] \ldots \;\Big\|\; y := \ldots T[i] \ldots \;\Big\|\; z := \ldots T[i] \ldots$$

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.

$$x := \dots T[i] \dots \;\Big\|\; y := \dots T[i] \dots \;\Big\|\; z := \dots T[i] \dots$$

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.

$$\frac{\{\, P_1 \,\}\; c_1 \;\{\, \lambda\_.\; Q_1 \,\} \qquad \{\, P_2 \,\}\; c_2 \;\{\, \lambda\_.\; Q_2 \,\}}{\{\, P_1 \star P_2 \,\}\; c_1 \parallel c_2 \;\{\, \lambda\_.\; Q_1 \star Q_2 \,\}}$$

- Cf. the “no race conditions” theorem in lecture #4.

A permission model

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

Naive idea: distinguish two permissions

Full permission:	$\ell \overset{1}{\mapsto} v$	(read, write, free)
Read-only permission:	$\ell \overset{R}{\mapsto} v$	(read)

A permission model

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

$$
\begin{array}{ccl}
\{\,\mathtt{emp}\,\} & \mathtt{alloc}(N) & \{\,\lambda\ell.\,\ell \overset{1}{\mapsto} \_ \star \cdots \star \ell + N - 1 \overset{1}{\mapsto} \_\,\} \\
\{\,[\![a]\!] \overset{\pi}{\mapsto} x\,\} & \mathtt{get}(a) & \{\,\lambda v.\,\langle v = x\rangle \star [\![a]\!] \overset{\pi}{\mapsto} x\,\} \quad (\pi \in \{1, R\}) \\
\{\,[\![a]\!] \overset{1}{\mapsto} \_\,\} & \mathtt{set}(a, a') & \{\,\lambda v.\,[\![a]\!] \overset{1}{\mapsto} [\![a']\!]\,\} \\
\{\,[\![a]\!] \overset{1}{\mapsto} \_\,\} & \mathtt{free}(a) & \{\,\lambda v.\,\mathtt{emp}\,\}
\end{array}
$$

A full permission can be weakened:

$$\ell \overset{1}{\mapsto} v \;\Rightarrow\; \ell \overset{R}{\mapsto} v$$

A read-only permission can be duplicated:

$$\ell \overset{R}{\mapsto} v \;=\; \ell \overset{R}{\mapsto} v \star \ell \overset{R}{\mapsto} v$$

Example of read-only sharing

$$\texttt{let}\ t = \texttt{alloc}(1)\ \texttt{in}$$
$$\texttt{set}(t, f(x));$$
$$\{\, t \overset{1}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \star t \overset{R}{\mapsto} f(x) \,\}$$

$$\begin{array}{l||l}
\{\, t \overset{R}{\mapsto} f(x) \,\} & \{\, t \overset{R}{\mapsto} f(x) \,\} \\
\quad \ldots \texttt{get}(t) \ldots & \quad \ldots \texttt{get}(t) \ldots \\
\quad \ldots & \quad \ldots \\
\{\, t \overset{R}{\mapsto} f(x) \star Q_1 \,\} & \{\, t \overset{R}{\mapsto} f(x) \star Q_2 \,\}
\end{array}$$

$$\{\, t \overset{R}{\mapsto} f(x) \star t \overset{R}{\mapsto} f(x) \star Q_1 \star Q_2 \,\}$$

Example of read-only sharing

$$\texttt{let}\ t = \texttt{alloc}(1)\ \texttt{in}$$
$$\texttt{set}(t, f(x));$$
$$\{\, t \overset{1}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \star t \overset{R}{\mapsto} f(x) \,\}$$

$$\begin{array}{l|l} \{\, t \overset{R}{\mapsto} f(x) \,\} & \{\, t \overset{R}{\mapsto} f(x) \,\} \\ \quad \ldots \texttt{get}(t) \ldots & \quad \ldots \texttt{get}(t) \ldots \\ \quad \ldots & \quad \ldots \\ \{\, t \overset{R}{\mapsto} f(x) \star Q_1 \,\} & \{\, t \overset{R}{\mapsto} f(x) \star Q_2 \,\} \end{array}$$

$$\{\, t \overset{R}{\mapsto} f(x) \star t \overset{R}{\mapsto} f(x) \star Q_1 \star Q_2 \,\}$$

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

Fractional permissions

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

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

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

The law for splitting and recombining permissions:

$$\ell \xmapsto{\pi+\pi'} v = \ell \xmapsto{\pi} v \ast \ell \xmapsto{\pi'} v \qquad \text{if } \pi + \pi' \in (0, 1]$$

Read-only sharing using fractional permissions

$$\begin{array}{l} \texttt{let}\ t = \texttt{alloc}(1)\ \texttt{in} \\ \texttt{set}(t, f(x)); \end{array}$$

$$\{\, t \overset{1}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{1/2}{\mapsto} f(x) \ast t \overset{1/2}{\mapsto} f(x) \,\}$$

$$\begin{array}{l} \{\, t \overset{1/2}{\mapsto} f(x) \,\} \\ \quad \ldots \texttt{get}(t) \ldots \\ \quad \ldots \\ \{\, t \overset{1/2}{\mapsto} f(x) \ast Q_1 \,\} \end{array} \;\Bigg\|\; \begin{array}{l} \{\, t \overset{1/2}{\mapsto} f(x) \,\} \\ \quad \ldots \texttt{get}(t) \ldots \\ \quad \ldots \\ \{\, t \overset{1/2}{\mapsto} f(x) \ast Q_2 \,\} \end{array}$$

$$\{\, t \overset{1/2}{\mapsto} f(x) \ast t \overset{1/2}{\mapsto} f(x) \ast Q_1 \ast Q_2 \,\} \Rightarrow \{\, t \overset{1}{\mapsto} f(x) \ast Q_1 \ast Q_2 \,\}$$

$$\texttt{free}(t)$$

$$\{\, Q_1 \ast Q_2 \,\}$$

Permission algebras

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

$$\begin{aligned} \text{commutative} \qquad & \pi_1 \oplus \pi_2 = \pi_2 \oplus \pi_1 \\ \text{and associative} \qquad & (\pi_1 \oplus \pi_2) \oplus \pi_3 = \pi_1 \oplus (\pi_2 \oplus \pi_3). \end{aligned}$$

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

Heaps with permissions

A heap h is a finite function from locations to pairs (π, 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

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

The combination is defined by

$$Dom(h_1 \oplus h_2) = Dom(h_1) \cup Dom(h_2)$$

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

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 $\uplus$.

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

$$P \mathrel{-\!\!\star} Q = \lambda h.\ \exists h_1, h_2,\ h_2 = h \oplus h_1 \wedge P\ h_1 \wedge Q\ h_2$$

In particular, $\ell \overset{\pi_1}{\mapsto} v_1 \star \ell \overset{\pi_2}{\mapsto} v_2$ holds if and only if $\pi_1 \oplus \pi_2$ is defined, $v_1 = v_2$, and $\ell \overset{\pi_1 \oplus \pi_2}{\longmapsto} v_1$.

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 $\{\, \ell \overset{1/2}{\mapsto} v \,\}\; c\; \{\, \lambda\_.\, \ell \overset{1/2}{\mapsto} v' \,\}$ with $v' \neq v$.

By framing with $\ell \overset{1/2}{\mapsto} v$, we get

$$\{\, \ell \overset{1}{\mapsto} v \,\}\; c\; \{\, \lambda\_.\, \ell \overset{1/2}{\mapsto} v' \ast \ell \overset{1/2}{\mapsto} v \,\}$$

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 $\ell \overset{1/2}{\mapsto} \_$, we can derive

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

even if $v' \neq v$.

Read-only sharing with counting permissions

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

Permissions π are integers ≥ -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:

$$\ell \overset{n}{\mapsto} v \;=\; \ell \xmapsto{n+1} v \ast \ell \xmapsto{-1} v \qquad \text{if } n \geq 0$$

One writer, several readers (Courtois, Heymans, Parnas, 1972)

Readers

$\{\ \mathtt{emp}\ \}$

$P(read);$
$count := count + 1;$
$\mathtt{if}\ count = 1\ \mathtt{then}\ P(write);$
$V(read);$

$\{\ b \overset{-1}{\mapsto} \_\ \}$

read b

$\{\ b \overset{-1}{\mapsto} \_\ \}$

$P(read);$
$count := count - 1;$
$\mathtt{if}\ count = 0\ \mathtt{then}\ V(write);$
$V(read);$

$\{\ \mathtt{emp}\ \}$

Writer

$\{\ \mathtt{emp}\ \}$

$P(write);$

$\{\ b \overset{0}{\mapsto} \_\ \}$

write b

$\{\ b \overset{0}{\mapsto} \_\ \}$

$V(write);$

$\{\ \mathtt{emp}\ \}$

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

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

Ghost code

Auxiliary variables, ghost variables (reminder from lecture #2)

Two ways to facilitate writing specifications as Hoare triples.

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

$$\forall \alpha, \beta,\ \ \{\, \mathtt{x} = \alpha \land \mathtt{y} = \beta \,\}\ \mathtt{if\ x < y\ then\ x := y}\ \{\, \mathtt{x} = \mathsf{max}(\alpha, \beta) \,\}$$

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

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

Ghost code

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`

$$\{\, r \geq 0 \land \exists q,\ a = b \cdot q + r \,\}$$

  $r := r - b$

`done`

$$\{\, r = a \bmod b \,\}$$

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

Example: remainder of Euclidean division

$$\{ a \geq 0 \}$$

$r := a;$

$q := 0;$

while $r \geq b$ do

$$\{ r \geq 0 \wedge a = b \cdot q + r \}$$

  $q := q + 1;$

  $r := r - b$

done

$$\{ r = a \bmod b \}$$

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.

$\texttt{def}\ DFS\ r =$

$\quad \texttt{if MARK}[r] = 0\ \texttt{then begin}$

$\quad\quad \texttt{MARK}[r] := 1;$

$\quad\quad \texttt{for}\ i = 0\ \texttt{to ARITY}[r] - 1\ \texttt{do}\ DFS(\texttt{CHILD}[r][i])\ \texttt{done}$

$\quad \texttt{end}$

It is surprisingly hard to prove this code correct!

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

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

Example: a recursive graph traversal

We can then show the invariant

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

and conclude

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

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.

$$PR := \varepsilon;\ CO := \varepsilon;$$

```
while true do                 ||  while true do
   compute x;                 ||     let y = consume() in
   PR := PR · x               ||     CO := CO · y
   produce(x);                ||     use y
done                          ||  done
```

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

The puzzle: $1 + 1 = 2$?

$$\texttt{set}(n, 0);$$
$$\texttt{atomic}(\mathit{incr}(n)) \mathbin{\|} \texttt{atomic}(\mathit{incr}(n))$$

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

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

$$\texttt{set}(n, 0);$$
$$\texttt{set}(a, 0); \texttt{set}(b, 0);$$
$$\texttt{atomic}(\mathit{incr}(n); \mathit{incr}(a)) \;\Big\|\; \texttt{atomic}(\mathit{incr}(n); \mathit{incr}(b))$$

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

Tracing processes with ghost code

$$\texttt{set}(n, 0);$$

$$\texttt{set}(a, 0); \texttt{set}(b, 0);$$

$$\texttt{atomic}(\mathit{incr}(n); \mathit{incr}(a)) \;\Big\|\; \texttt{atomic}(\mathit{incr}(n); \mathit{incr}(b))$$

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:

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

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

$$J \;=\; \exists x, y,\ a \overset{1/2}{\mapsto} x \ast b \overset{1/2}{\mapsto} y \ast n \overset{1}{\mapsto} x + y$$

We have:

$$\{ J \ast a \overset{1/2}{\mapsto} x \} \Rightarrow$$

$$\{ a \overset{1}{\mapsto} x \ast \exists y, b \overset{1/2}{\mapsto} y \ast n \overset{1}{\mapsto} x + y \}$$

$incr(n);$

$incr(a);$

$$\{ a \overset{1}{\mapsto} x + 1 \ast \exists y, b \overset{1/2}{\mapsto} y \ast n \overset{1}{\mapsto} x + 1 + y \}$$

$$\Rightarrow \{ J \ast a \overset{1/2}{\mapsto} x + 1 \}$$

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

Fractional permissions to the rescue

We can then derive:

$$\texttt{set}(n,0);\texttt{set}(a,0);\texttt{set}(b,0);$$

$$\{\, n \overset{1}{\mapsto} 0 \ast a \overset{1}{\mapsto} 0 \ast b \overset{1}{\mapsto} 0 \,\} \Rightarrow \{\, J \ast a \overset{1/2}{\mapsto} 0 \ast b \overset{1/2}{\mapsto} 0 \,\}$$

$$\begin{array}{l|l} \{\, a \overset{1/2}{\mapsto} 0 \,\} & \{\, b \overset{1/2}{\mapsto} 0 \,\} \\ \quad \texttt{atomic}(incr(n); incr(a)) & \quad \texttt{atomic}(incr(n); incr(b)) \\ \{\, a \overset{1/2}{\mapsto} 1 \,\} & \{\, b \overset{1/2}{\mapsto} 1 \,\} \end{array}$$

$$\{\, J \ast a \overset{1/2}{\mapsto} 1 \ast b \overset{1/2}{\mapsto} 1 \,\} \Rightarrow \{\, n \overset{1}{\mapsto} 2 \ast a \overset{1}{\mapsto} 1 \ast b \overset{1}{\mapsto} 1 \,\}$$

Therefore, $n = 2$ in the end!

Fractional permissions to the rescue

We can then derive:

$$\texttt{set}(n,0); \texttt{set}(a,0); \texttt{set}(b,0);$$

$$\{\, n \overset{1}{\mapsto} 0 \ast a \overset{1}{\mapsto} 0 \ast b \overset{1}{\mapsto} 0 \,\} \Rightarrow \{\, J \ast a \overset{1/2}{\mapsto} 0 \ast b \overset{1/2}{\mapsto} 0 \,\}$$

$$\left.\begin{array}{l} \{\, a \overset{1/2}{\mapsto} 0 \,\} \\ \qquad \texttt{atomic}(\mathit{incr}(n); \mathit{incr}(a)) \\ \{\, a \overset{1/2}{\mapsto} 1 \,\} \end{array} \right\| \begin{array}{l} \{\, b \overset{1/2}{\mapsto} 0 \,\} \\ \qquad \texttt{atomic}(\mathit{incr}(n); \mathit{incr}(b)) \\ \{\, b \overset{1/2}{\mapsto} 1 \,\} \end{array}$$

$$\{\, J \ast a \overset{1/2}{\mapsto} 1 \ast b \overset{1/2}{\mapsto} 1 \,\} \Rightarrow \{\, n \overset{1}{\mapsto} 2 \ast a \overset{1}{\mapsto} 1 \ast b \overset{1}{\mapsto} 1 \,\}$$

Therefore, $n = 2$ in the end!

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

$\rightarrow$ seminar #5 by J. H. Jourdan.

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.

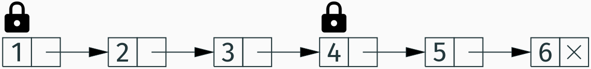
Example: one process removes “2”, the other removes “5”.

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”.

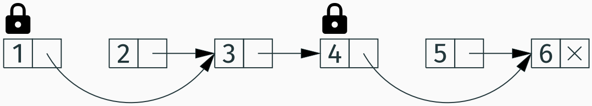


Example: singly-linked lists

```
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”.

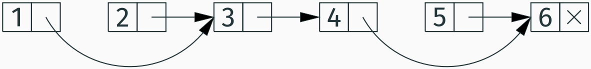


Example: singly-linked lists

```
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”.

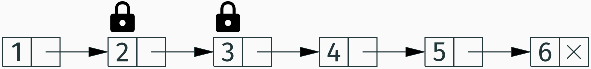
Locking one node at a time is not enough!
Example: one process removes “3”, the other removes “4”.

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”.



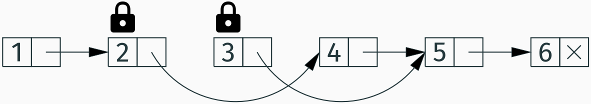
Locking one node at a time is not enough!
Example: one process removes “3”, the other removes “4”.

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

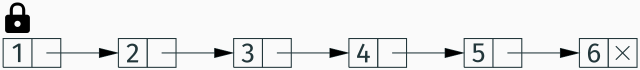
To modify a node, we must have locked the node as well as the node before.

Example: removal of “4”.

Hand-over-hand locking

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

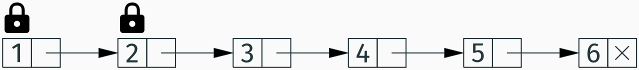
Example: removal of “4”.



Hand-over-hand locking

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

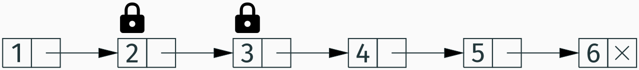
Example: removal of “4”.



Hand-over-hand locking

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

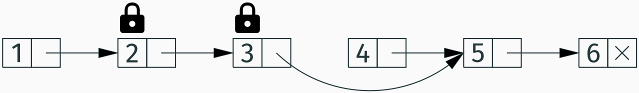
Example: removal of “4”.



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

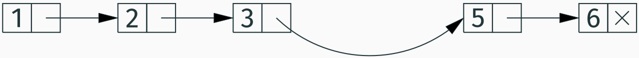
To modify a node, we must have locked the node as well as the node before.

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:

$\ell \overset{\pi}{\bullet\!\!\to} \mathit{RI}$ "at location ℓ, with permission π, there is a lock that protects the resource described by RI"

🔒 ℓ "the lock at location ℓ is locked by the current process"

The "small rules" for locks:

$$\{\,\ell \overset{\pi}{\bullet\!\!\to} \mathit{RI}\,\}\quad \texttt{lock}(\ell)\quad \{\,\ell \overset{\pi}{\bullet\!\!\to} \mathit{RI} \star 🔒\,\ell \star \mathit{RI}\,\}$$

$$\{\,\ell \overset{\pi}{\bullet\!\!\to} \mathit{RI} \star 🔒\,\ell \star \mathit{RI}\,\}\quad \texttt{unlock}(\ell)\quad \{\,\ell \overset{\pi}{\bullet\!\!\to} \mathit{RI}\,\}$$

$$\{\,\ell \overset{1}{\mapsto} \_ \star \mathit{RI}\,\}\quad \texttt{initlock}(\ell)\quad \{\,\ell \overset{1}{\bullet\!\!\to} \mathit{RI}\,\}$$

$$\{\,\ell \overset{1}{\bullet\!\!\to} \mathit{RI}\,\}\quad \texttt{destroylock}(\ell)\quad \{\,\ell \overset{1}{\mapsto} \_ \star \mathit{RI}\,\}$$

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.

$$\begin{aligned}
\mathit{list}(p, n) = {} & (\langle n = +\infty \rangle \ast p.\mathtt{val} \overset{1}{\mapsto} n \ast p.\mathtt{next} \overset{1}{\mapsto} \mathtt{NULL}) \\
& \vee (\exists q, n',\ \langle n < n' \rangle \ast p.\mathtt{val} \overset{1}{\mapsto} n \ast p.\mathtt{next} \overset{1}{\mapsto} q \\
& \qquad \ast\, p.\mathtt{lock} \overset{1}{\bullet\!\!\rightarrow} \mathit{list}(q, n'))
\end{aligned}$$

$$\begin{aligned}
\mathit{listhead}(p, \pi) = {} & \exists q, n,\ p.\mathtt{val} \overset{\pi}{\mapsto} -\infty \ast p.\mathtt{next} \overset{\pi}{\mapsto} q \\
& \ast\, p.\mathtt{lock} \overset{\pi}{\bullet\!\!\rightarrow} \mathit{list}(q, n)
\end{aligned}$$

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

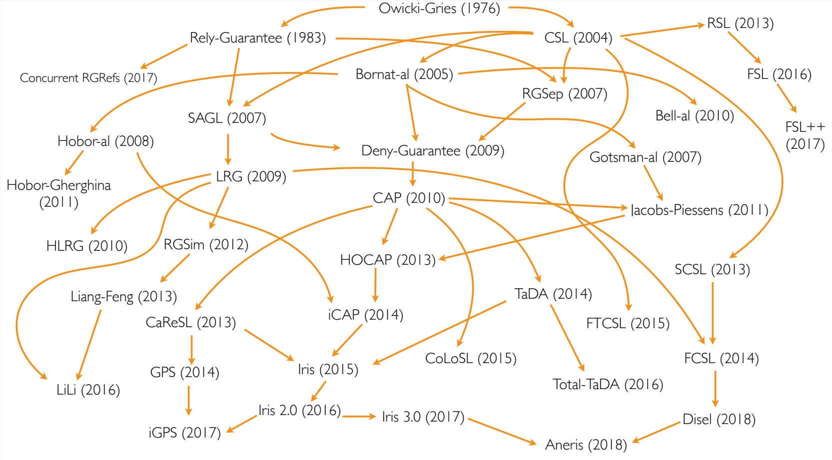
Summary

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 $\mathtt{x} = \alpha$ (in Hoare logic)
- Facts about the memory heap $\mathtt{emp}$, $\ell \mapsto v$, $\ell \mapsto \_$
- The same plus permissions $\ell \overset{\pi}{\mapsto} v$
- Facts about locks $\ell \overset{\pi}{\bullet\!\!\rightarrow} RI$, 🔒 ℓ
- Facts about ghost states.
- Time credits (→ seminar by F. Pottier)
- Transition systems (→ seminar by J. H. Jourdan)
- 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

References

All about Iris:

- J.-H. Jourdan's seminar (#6).
- 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.