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”  \( \{ \ell \mapsto \_ \} \) \( \text{set}(\ell, v) \)  \( \{ \lambda. \ell \mapsto v \} \)

- 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

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

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 way to adapt what we already know \( \{ P' \} c \{ Q' \} \) to what we need to prove \( \{ P \} c \{ Q \} \).

\[
\begin{align*}
\{ P' \} c \{ Q' \} & \quad P \Rightarrow P' \star R \quad \forall v, Q' v \star R \Rightarrow Q v \\
\hline
\{ P \} c \{ Q \}
\end{align*}
\]

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 \( R \).
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:

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

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

$$P \Rightarrow Q \overset{\text{def}}{=} \lambda h. \forall h' \perp h, P h' \Rightarrow Q(h' \cup h)$$
Magic wand = separating implication

Separating implication $\rightsquigarrow$ is to separating conjunction $\star$ what plain implication $\Rightarrow$ is to plain conjunction $\land$.

Adjunction property:

$$H \Rightarrow (P \rightsquigarrow Q) \iff H \star P \Rightarrow Q$$

Some other properties:

$$P \star (P \rightarrow \star Q) \Rightarrow Q \quad \text{(elimination)}$$

$$\text{emp} \Rightarrow P \rightarrow \star P \quad \text{(idempotence)}$$

$$(P \rightarrow \star Q) \star (Q \rightarrow \star R) \Rightarrow P \rightarrow \star R \quad \text{(transitivity)}$$

$$(P \star Q) \rightarrow \star R = P \star Q \rightarrow \star R \quad \text{(currying)}$$

$$(P \star Q) \star R \Rightarrow P \rightarrow \star (Q \star R) \quad \text{(distribution)}$$
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:

\[ \{ P \} \ c \ \{ Q \} \ \text{if and only if} \ \ P \ \Rightarrow \ wp \ c \ Q \]

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 \( Q \)?”
The rules of separation logic can be rephrased using $wp$:

$$Q \Box[a] \Rightarrow wp\ a\ Q$$

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

$$(\text{if } \Box[b] \text{ then } wp\ c_1\ Q\ \text{ else } wp\ c_2\ Q) \Rightarrow wp\ (\text{if } b \text{ then } c_1\ \text{ else } c_2)\ Q$$
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.

\[
\begin{align*}
\text{emp} & \Rightarrow \ wp (\text{alloc}(N)) (\lambda \ell. \ell \mapsto _\ast \cdots \ast \ell + N - 1 \mapsto _\ast) \\
\llbracket a \rrbracket \mapsto x & \Rightarrow \ wp (\text{get}(a)) (\lambda v. \langle v = x \rangle \ast \llbracket a \rrbracket \mapsto x) \\
\llbracket a \rrbracket \mapsto - & \Rightarrow \ wp (\text{set}(a, a')) (\lambda v. \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket) \\
\llbracket a \rrbracket \mapsto - & \Rightarrow \ wp (\text{free}(a)) (\lambda v. \text{emp})
\end{align*}
\]

Time to wave the magic wand...
Structural rules for weakest preconditions

The frame rule:

\[(\text{wp } c \ Q) \star R \Rightarrow \text{wp } c \ (\lambda v. \ Q \ v \star R)\]

The consequence rule:

\[
\forall v, \ Q \ v \Rightarrow Q' \ v
\]

\[
\frac{\text{wp } c \ Q \Rightarrow wp c \ Q'}{\text{wp } c \ Q' \Rightarrow wp c \ Q'}
\]

The ramified consequence rule:

\[wp c \ Q \star (\forall v, \ Q \ v \rightarrow \star Q' \ v) \Rightarrow wp c \ Q'\]
Applying ramification to the \( wp \) calculation for imperative constructs, we obtain \( wp \) equations usable for any postcondition \( Q \).

\[
\forall \ell, \ (\ell \mapsto \_ \star \cdots \star \ell + N - 1 \mapsto \_ \star Q \ell) \Rightarrow wp(alloc(N)) Q
\]

\[
\exists x, \ [a] \mapsto x \star ([a] \mapsto x \star Q x) \Rightarrow wp(get(a)) Q
\]

\[
[a] \mapsto \_ \star (\forall v, \ [a] \mapsto [a'] \star Q v) \Rightarrow wp(set(a, a')) Q
\]

\[
([a] \mapsto \_) \star (\forall v, Q v) \Rightarrow wp(free(a)) Q
\]
A taste of the “Iris proof mode”

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

\[
\begin{array}{c}
\vdash x_1 \ldots x_n \ H_1 \ldots H_m \\
\hline
P
\end{array}
\]

instead of formulas \( \forall x_1, \ldots x_n, H_1 \land \cdots \land H_m \Rightarrow P \).
A taste of the “Iris proof mode”

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

\[ x_1 \ldots x_n \ H_1 \ldots H_m \]

\[ \vdash P \]

instead of formulas \( \forall x_1, \ldots x_n, H_1 \land \cdots \land H_m \Rightarrow P \).

A proof context in separation logic is:

\[ x_1 \ldots x_n \ H_1 \ldots H_m \quad ({\text{standard hypotheses}}) \]

\[ \vdash P_1 \ldots P_k \quad ({\text{spatial hypotheses}}) \]

\[ \vdash Q \quad ({\text{goal}}) \]

It stands for \( \forall x_i, H_1 \land \cdots \land H_m \Rightarrow P_1 \SBL P_k \Rightarrow Q \).
Some introduction rules

\[
\begin{align*}
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
P_1 \star \cdots \star P_n \rightarrow Q
\end{align*}
\]

\[
\begin{align*}
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\vdots & \quad \vdots \\
\quad & \quad \\
\quad & \quad \\
\exists x, P x
\end{align*}
\]
Weakest preconditions $\approx$ symbolic execution

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

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

\[
\text{wp } (\text{let } x = c_1 \text{ in } c_2) \ Q \Rightarrow \cdots \Rightarrow \text{wp } c_1 (\lambda v. \text{wp } c_2[x \leftarrow v] \ Q) \Rightarrow \cdots \Rightarrow \text{wp } a (\lambda v. \text{wp } c_2[x \leftarrow v] \ Q) \Rightarrow \cdots \Rightarrow \text{wp } c_2[x \leftarrow a] \ Q
\]
Symbolic execution of memory operations

The \textit{wp} rules for the memory operations become clearer:

\[
\exists x, \ [a] \mapsto x \star ([a] \mapsto x \star Q x) \Rightarrow wp (get(a)) Q
\]

\[
\vdots \quad [a] \mapsto x \quad \vdots \quad \mapsto \quad wp (get(a)) Q
\]

\[
[a] \mapsto _\star (\forall v, [a] \mapsto [a'] \star Q v) \Rightarrow wp (set(a, a')) Q
\]

\[
\vdots \quad [a] \mapsto _\star \quad \vdots \quad \mapsto \quad wp (set(a, a')) Q
\]

16
Symbolic execution of memory operations

$$\forall \ell, (\ell \mapsto \_ \blacksquare \ldots \blacksquare \ell + N - 1 \mapsto \_) \mapsto Q \ell \Rightarrow wp(\text{alloc}(N)) Q$$

$$\ldots$$

$$\left\lfloor a \right\rfloor \mapsto \_ \quad \ldots$$

$$wp(\text{alloc}(N)) Q$$

$$\ldots \quad \ell \mapsto \_ \quad \ldots \quad \ell + N - 1 \mapsto \_$$

$$Q \ell$$

$$(\left\lfloor a \right\rfloor \mapsto \_ \blacklozenge (\forall v, Q v) \Rightarrow wp(\text{free}(a)) Q$$

$$\ldots$$

$$\left\lfloor a \right\rfloor \mapsto \_ \quad \ldots \quad \left\lfloor a \right\rfloor \mapsto \_ \quad \ldots$$

$$wp(\text{free}(a)) Q$$

$$\ldots \quad v$$

$$Q v$$
Partial permissions
Several processes access a shared data structure without synchronization, but do not modify the data structure.

\[ x := \ldots T[i] \ldots \parallel y := \ldots T[i] \ldots \parallel z := \ldots T[i] \ldots \]

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

\[ x := \ldots T[i] \ldots \parallel y := \ldots T[i] \ldots \parallel z := \ldots T[i] \ldots \]

This is efficient:

- No need to copy the data structure in each process. (≠ distributed-memory parallelism).
Read-only sharing

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

\[ x := \ldots T[i] \ldots \parallel y := \ldots T[i] \ldots \parallel z := \ldots T[i] \ldots \]

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.

\[
\{ P_1 \} \ c_1 \ \{ \lambda_. \ Q_1 \} \ \{ P_2 \} \ c_2 \ \{ \lambda_. \ Q_2 \}
\]

\[
\{ P_1 \parallel P_2 \} \ c_1 \parallel c_2 \ \{ \lambda_. \ Q_1 \parallel Q_2 \}
\]

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

The assertion $\ell \rightarrow v$, “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: $\ell \xleftarrow{1} V$ (read, write, free)

Read-only permission: $\ell \xleftarrow{R} V$ (read)
A permission model

The “small rule” for \texttt{get} accepts both permissions. The other small rules produce or require full permissions.

\[
\begin{align*}
\{ \text{emp} \} \quad & \text{alloc}(N) \quad \{ \lambda \ell. \ell \mapsto 1 \star \cdots \star \ell + N - 1 \mapsto 1 \} \\
\{ \llbracket a \rrbracket \xrightarrow{\pi} x \} \quad & \text{get}(a) \quad \{ \lambda v. \langle v = x \rangle \star \llbracket a \rrbracket \xrightarrow{\pi} x \} \quad (\pi \in \{1, R\}) \\
\{ \llbracket a \rrbracket \mapsto 1 \} \quad & \text{set}(a, a') \quad \{ \lambda v. \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \} \\
\{ \llbracket a \rrbracket \mapsto 1 \} \quad & \text{free}(a) \quad \{ \lambda v. \text{emp} \}
\end{align*}
\]

A full permission can be weakened:

\[
\ell \mapsto 1 \Rightarrow \ell \mapsto R \Rightarrow \ell \mapsto v
\]

A read-only permission can be duplicated:

\[
\ell \mapsto R \Rightarrow v = \ell \mapsto R \Rightarrow v \star \ell \mapsto R \Rightarrow v
\]
Example of read-only sharing

let \( t = \text{alloc}(1) \) in

\[
\text{set}(t, f(x));
\]

\[
\{ t \mapsto f(x) \} \Rightarrow \{ t \mapsto^R f(x) \} \Rightarrow \{ t \mapsto^R f(x) \star t \mapsto^R f(x) \}
\]

\[
\{ t \mapsto^R f(x) \} \quad \| \quad \{ t \mapsto^R f(x) \}
\]

... get(t) ...

... get(t) ...

... 

\[
\{ t \mapsto^R f(x) \star Q_1 \} \quad \| \quad \{ t \mapsto^R f(x) \star Q_2 \}
\]

\[
\{ t \mapsto^R f(x) \star t \mapsto^R f(x) \star Q_1 \star Q_2 \}
\]

Problem: we cannot free \( t \) at the end of this code!
Example of read-only sharing

```
let t = alloc(1) in
set(t, f(x));
{ t \mapsto f(x) } \Rightarrow \{ t \mapsto f(x) \} \Rightarrow \{ t \mapsto f(x) \ast t \mapsto f(x) \}

\{ t \mapsto f(x) \} \quad \| \quad \{ t \mapsto f(x) \}
\quad \ldots \text{get}(t) \ldots \quad \| \quad \ldots \text{get}(t) \ldots
\quad \ldots \quad \| \quad \ldots
\quad \{ t \mapsto f(x) \ast Q_1 \} \quad \| \quad \{ t \mapsto f(x) \ast Q_2 \}

\{ t \mapsto f(x) \ast t \mapsto f(x) \ast Q_1 \ast 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:

\[
\ell \overset{\pi + \pi'}{\longrightarrow} v = \ell \overset{\pi}{\longrightarrow} v \ast \ell \overset{\pi'}{\longrightarrow} v \quad \text{if} \quad \pi + \pi' \in (0, 1]
\]
let \( t = \text{alloc}(1) \) in
\[
\text{set}(t, f(x));
\]
\[
\{ t \mapsto f(x) \} \Rightarrow \{ t \mapsto f(x) \ast t \mapsto f(x) \}
\]
\[
\{ t \mapsto f(x) \ast Q_1 \} \Rightarrow \{ t \mapsto f(x) \ast Q_2 \}
\]
\[
\{ t \mapsto f(x) \ast t \mapsto f(x) \ast Q_1 \ast Q_2 \} \Rightarrow \{ t \mapsto f(x) \ast Q_1 \ast Q_2 \}
\]
\[
\text{free}(t)
\]
\[
\{ Q_1 \ast Q_2 \}
\]
A set $\Pi$ equipped with a partial operation $\oplus$ to combine two permissions. The operation is

- **commutative**
  \[ \pi_1 \oplus \pi_2 = \pi_2 \oplus \pi_1 \]

- **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 $\pi$”.
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:

$$(\pi_1, v_1) \oplus (\pi_2, v_2) = (\pi_1 \oplus \pi_2, v_1) \text{ if } v_1 = v_2 \text{ and } \pi_1 \oplus \pi_2 \text{ is defined}$$
The combination $h_1 \oplus h_2$ of two heaps is defined if

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

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 $h_1 \uplus h_2$. 
The usual definitions, using heap combination $\oplus$ instead of disjoint union $\uplus$.

\[ P \star Q = \lambda h. \exists h_1, h_2, \ h = h_1 \oplus h_2 \land P h_1 \land Q h_2 \]

\[ P \rightarrow\star Q = \lambda h. \exists h_1, h_2, \ h_2 = h \oplus h_1 \land P h_1 \land Q h_2 \]

In particular, $\ell \overset{\pi_1}{\rightarrow} v_1 \star \ell \overset{\pi_2}{\rightarrow} v_2$ holds if and only if $\pi_1 \oplus \pi_2$ is defined, $v_1 = v_2$, and $\ell \overset{\pi_1 \oplus \pi_2}{\rightarrow} v_1$. 
An interesting property: every command $c$ provable with a precondition “read-only permission on location $\ell$” cannot modify $\ell$.

Proof sketch in the sequential case:

By way of contradiction, assume $\{ \ell \xrightarrow{1/2} v \} \mathsf{c} \{ \lambda_. \ell \xrightarrow{1/2} v' \}$ with $v' \neq v$.

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

$$\{ \ell \xrightarrow{1} v \} \mathsf{c} \{ \lambda_. \ell \xrightarrow{1/2} v' \ast \ell \xrightarrow{1/2} v \}$$

The precondition is true, but the postcondition is always false, since location $\ell$ 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 $\ell \overset{\frac{1}{2}}{\rightarrow} \_\,$, we can derive

$$\ell \overset{\frac{1}{2}}{\rightarrow} \_ \vdash \{ \ell \overset{\frac{1}{2}}{\rightarrow} v \} \text{atomic(set}(\ell, v')) \{ \lambda_. \ell \overset{\frac{1}{2}}{\rightarrow} v' \}$$

even if $v' \neq v$. 
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:

$$\ell \xrightarrow{n} v = \ell \xrightarrow{n+1} v \star \ell \xrightarrow{-1} v \quad \text{if } n \geq 0$$
One writer, several readers

(Courtois, Heymans, Parnas, 1972)

Readers

\[ \{ \text{emp} \} \]

\[ P(\text{read}); \]
\[ \text{count} := \text{count} + 1; \]
\[ \text{if count} = 1 \text{ then } P(\text{write}); \]
\[ V(\text{read}); \]

\[ \{ b \xrightarrow{-1} \_ \} \]

\[ \text{read } b \]

\[ \{ b \xrightarrow{-1} \_ \} \]

\[ P(\text{read}); \]
\[ \text{count} := \text{count} - 1; \]
\[ \text{if count} = 0 \text{ then } V(\text{write}); \]
\[ V(\text{read}); \]

\[ \{ \text{emp} \} \]

Writer

\[ \{ \text{emp} \} \]

\[ P(\text{write}); \]

\[ \{ b \xrightarrow{0} \_ \} \]

\[ \text{write } b \]

\[ \{ b \xrightarrow{0} \_ \} \]

\[ V(\text{write}); \]

\[ \{ \text{emp} \} \]

Invariant for write: \[ b \xrightarrow{0} \_ \]

Invariant for read: \[ \exists n, \text{count} \xrightarrow{0} n \ast (\langle n = 0 \rangle \lor \langle n > 0 \rangle \ast b \xrightarrow{n} \_ ) \]
Ghost code
Two ways to facilitate writing specifications as Hoare triples.

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

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

**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; \\
\\nwhile r \geq b do \\
\{ r \geq 0 \land \exists q, a = b \cdot q + r \} \\
\r := r - b \\
done \\
\{ r = a \mod b \} \\

Automated theorem provers sometimes have a hard time with existential quantification...
Example: remainder of Euclidean division

\[ \{ a \geq 0 \} \]

\( r := a; \)
\( \mathcal{G} \ q := 0; \)
\( \text{while } r \geq b \text{ do} \)
\( \{ r \geq 0 \land a = b \cdot q + r \} \)
\( \mathcal{G} \ q := q + 1; \)
\( r := r - b \)
\( \text{done} \)
\( \{ r = a \mod b \} \)

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 $r$.

```python
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. \((W \approx \text{the nodes that remain to be traversed})\)

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

We can then show the invariant

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

and conclude

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

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.

\[ PR := \varepsilon; \quad CO := \varepsilon; \]

while true do
  compute \( x \);
  \( PR := PR \cdot x \)
  produce(\( x \));
  done

while true do
  let \( y = consume() \) in
  \( CO := CO \cdot y \)
  use \( y \)
  done

The ghost lists \( PR \) and \( CO \) keep track of the data produced or consumed.
The puzzle: $1 + 1 = 2$?

\[
\text{set}(n, 0); \\
\text{atomic}(\text{incr}(n)) \parallel \text{atomic}(\text{incr}(n))
\]

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

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

\[ \text{set}(n, 0); \]
\[ \text{set}(a, 0); \text{set}(b, 0); \]
\[ \text{atomic}(\text{incr}(n); \text{incr}(a)) \parallel \text{atomic}(\text{incr}(n); \text{incr}(b)) \]

\( a \) represents the contribution of the left process to the sum \( n \),
\( b \) represents that of the right process.
Tracing processes with ghost code

\[
\begin{align*}
\text{set}(n, 0); \\
\text{atomic}(\text{incr}(n); \text{set}(a, 0); \text{set}(b, 0)); \\
\text{atomic}(\text{incr}(n); \text{incr}(a)) \parallel \text{atomic}(\text{incr}(n); \text{incr}(b))
\end{align*}
\]

\(a\) 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.
Consider the resource invariant

\[ J = \exists x, y, a^{\frac{1}{2}} \rightarrow x \ast b^{\frac{1}{2}} \rightarrow y \ast n^{1} \rightarrow x + y \]

We have:

\[ \{ J \ast a^{\frac{1}{2}} \rightarrow x \} \Rightarrow \]

\[ \{ a^{1} \rightarrow x \ast \exists y, b^{\frac{1}{2}} \rightarrow y \ast n^{1} \rightarrow x + y \} \]

\text{incr}(n);
\text{incr}(a);

\[ \{ a^{1} \rightarrow x + 1 \ast \exists y, b^{\frac{1}{2}} \rightarrow y \ast n^{1} \rightarrow x + 1 + y \} \Rightarrow \{ J \ast a^{\frac{1}{2}} \rightarrow x + 1 \} \]

Therefore, \( J \vdash \{ a^{\frac{1}{2}} \rightarrow x \} \) \text{atomic(incr}(n); incr(a)) \{ a^{\frac{1}{2}} \rightarrow x + 1 \} \]
Fractional permissions to the rescue

We can then derive:

\[ \text{set}(n, 0); \text{set}(a, 0); \text{set}(b, 0); \]
\[ \{ n \overset{1}{\rightarrow} 0 \ast a \overset{1}{\rightarrow} 0 \ast b \overset{1}{\rightarrow} 0 \} \Rightarrow \{ J \ast a \overset{1/2}{\rightarrow} 0 \ast b \overset{1/2}{\rightarrow} 0 \} \]
\[ \{ a \overset{1/2}{\rightarrow} 0 \} \]
\[ \text{atomic}(\text{incr}(n); \text{incr}(a)) \]
\[ \{ a \overset{1/2}{\rightarrow} 1 \} \]
\[ \{ J \ast a \overset{1/2}{\rightarrow} 1 \ast b \overset{1/2}{\rightarrow} 1 \} \Rightarrow \{ n \overset{1}{\rightarrow} 2 \ast a \overset{1}{\rightarrow} 1 \ast b \overset{1}{\rightarrow} 1 \} \]

Therefore, \( n = 2 \) in the end!
We can then derive:

\[
\text{set}(n, 0); \text{set}(a, 0); \text{set}(b, 0);
\]

\[
\{ n \mapsto 0 \star a \mapsto 0 \star b \mapsto 0 \} \Rightarrow \{ J \star a \mapsto^{1/2} 0 \star b \mapsto^{1/2} 0 \}
\]

\[
\{ a \mapsto^{1/2} 0 \}
\]

atomic(\text{incr}(n); \text{incr}(a)) \quad \mid \quad \text{atomic(\text{incr}(n); \text{incr}(b))}

\[
\{ a \mapsto^{1/2} 1 \}
\]

\[
\{ n \mapsto 2 \star a \mapsto 1 \star b \mapsto 1 \}
\]

Therefore, \( n = 2 \) in the end!

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

→ seminar #5 by J. H. Jourdan.
Storable locks
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

```c
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”.

```
| 1 | 2 | 3 | 4 | 5 | 6 |
```

×
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”.
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

```c
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”.

![Diagram of singly-linked list with nodes 1 to 6 and arrows indicating connections and order of removal.]
Example: singly-linked lists

```c
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”.

![Diagram showing linked list with nodes 1, 2, 3, 4, 5, 6, and an X symbol replacing node 6]

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

```c
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”.

[Diagram showing locked nodes]

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

```c
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”.

```
1 - 2  3 - 4 - 5 - 6 x
```

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”.

1 → 2 → 3 → 4 → 5 → 6
To modify a node, we must have locked the node as well as the node before.

Example: removal of “4”.

\[\begin{array}{c}
\text{1} \rightarrow \text{2} \rightarrow \text{3} \rightarrow \text{4} \rightarrow \text{5} \rightarrow \text{6} \times \\
\end{array}\]
Hand-over-hand locking

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

Example: removal of “4”.

1 - 2 - 3 - 4 - 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”.

1 → 2 → 3 → 4 → 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”.

1 → 2 → 3 → 4 → 5 → 6 ×
To modify a node, we must have locked the node as well as the node before.

Example: removal of “4”.

1 → 2 → 3 → 4 → 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”.

1 → 2 → 3 → 5 → 6 ×
Specification of stored locks

Two new assertions:

\[ \ell \xrightarrow{\pi} RI \quad \text{“at location } \ell \text{, with permission } \pi \text{, there is a lock that protects the resource described by } RI \text{”} \]

\[ \bullet \! \ell \quad \text{“the lock at location } \ell \text{ is locked by the current process”} \]

The “small rules” for locks:

\[
\begin{align*}
\{ \ell \xrightarrow{\pi} RI \} & \quad \text{lock}(\ell) & \{ \ell \xrightarrow{\pi} RI \star \bullet \ell \star RI \} \\
\{ \ell \xrightarrow{\pi} RI \star \bullet \ell \star RI \} & \quad \text{unlock}(\ell) & \{ \ell \xrightarrow{\pi} RI \} \\
\{ \ell \xrightarrow{1} _{-} \star RI \} & \quad \text{initlock}(\ell) & \{ \ell \xrightarrow{1} RI \} \\
\{ \ell \xrightarrow{1} RI \} & \quad \text{destroylock}(\ell) & \{ \ell \xrightarrow{1} _{-} \star RI \}
\end{align*}
\]
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.

$$list(p, n) = (\langle n = +\infty \rangle \star p.val \xrightarrow{1} n \star p.next \xrightarrow{1} \text{NULL})$$

$$\lor (\exists q, n', \langle n < n' \rangle \star p.val \xrightarrow{1} n \star p.next \xrightarrow{1} q$$

$$\star p.lock \xrightarrow{1} list(q, n'))$$

$$listhead(p, \pi) = \exists q, n, \quad p.val \xrightarrow{\pi} -\infty \star p.next \xrightarrow{\pi} q$$

$$\star p.lock \xrightarrow{\pi} list(q, n)$$

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 \( \text{emp}, l \rightarrow v, l \rightarrow \_ \)
• The same plus permissions \( l \xrightarrow{\pi} v \)
• Facts about locks \( l \xrightarrow{\pi} RI, \lock l \)
• Facts about ghost states.
• Time credits (\( \rightarrow \) seminar by F. Pottier)
• Transition systems (\( \rightarrow \) 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:


Storable locks: