Program logics, third lecture

Pointers and data structures: separation logic

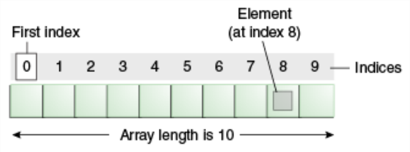
Xavier Leroy

2021-03-18

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr

Prologue:
arrays in Hoare logic

Arrays



Adding arrays to IMP

Expressions: $a ::= \ldots \mid T[a]$ reading from array T

Commands: $c ::= \ldots \mid T[a] := a'$ writing to array T

Convention: uppercase variables T, U, are arrays.

A Hoare logic for arrays

Which rule for array writes?

Wrong: $\{\, Q[T[a] \leftarrow a'] \,\}\ T[a] := a'\ \{\, Q \,\}$ ✘

It's not just $T[a]$ that is modified, but also $T[a_1]$ for every expression a_1 that has the same value as a. Example:

$$\{\, 0 = 0 \wedge T[\,i\,] = 1 \,\}\ T[0] := 0\ \{\, T[0] = 0 \wedge T[\,i\,] = 1 \,\}$$

is false if $i = 0$.

Correct: $\{\, Q[T \leftarrow (T + a \mapsto a')] \,\}\ T[a] := a'\ \{\, Q \,\}$ ✔

The expression $T + a \mapsto a'$ denotes a functional update: an array equal to T except that index a has value a'.

Reasoning about arrays

We reason about these functional updates with the equation

$$(T + a \mapsto a')\,[\,i\,] = \begin{cases} a' & \text{if } \; i = a \\ T[\,i\,] & \text{if } \; i \neq a \end{cases}$$

Example (verifying a write to `T[0]`)

$$\{\, i \neq 0 \land \mathtt{T}[\,i\,] = 1 \,\} \iff$$
$$\{\, 0 = 0 \land (i = 0 \,?\, 0 : \mathtt{T}[\,i\,]) = 1 \,\} \iff$$
$$\{\, (\mathtt{T} + 0 \mapsto 0)[0] = 0 \land (\mathtt{T} + 0 \mapsto 0)[\,i\,] = 1 \,\}$$

$\mathtt{T}[0] := 0$

$$\{\, \mathtt{T}[0] = 0 \land \mathtt{T}[\,i\,] = 1 \,\}$$

Example: array initialization

$i := 0;$

$$\{\, i = 0 \,\}$$

$\texttt{while } i < N \texttt{ do}$

$$\{\, \forall j,\ 0 \leq j < i \Rightarrow \texttt{T}[j] = j \times 2 \,\} \Rightarrow$$

$$\{\, \forall j,\ 0 \leq j < i + 1 \Rightarrow (\texttt{T} + i \mapsto i \times 2)[j] = j \times 2 \,\}$$

$\quad \texttt{T}[i] := i \times 2;$

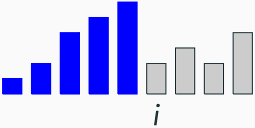
$$\{\, \forall j,\ 0 \leq j < i + 1 \Rightarrow \texttt{T}[j] = j \times 2 \,\}$$

$\quad i := i + 1$

$$\{\, \forall j,\ 0 \leq j < i \Rightarrow \texttt{T}[j] = j \times 2 \,\}$$

$\texttt{done}$

Example: insertion sort



$i := 1;$

while $i < N$ do

$\quad\quad \{\, 0 < i < N \;\; \wedge \;\; \forall p, q,\; 0 \leq p \leq q < i \Rightarrow \mathtt{T}[p] \leq \mathtt{T}[q] \,\}$

$\quad j := i;$

$\quad$ while $j > 0 \wedge T[j-1] > T[j]$

$\quad\quad \{\, 0 \leq j \leq i \;\; \wedge \;\; \forall p, q,\; 0 \leq p \leq q \leq i \wedge q \neq j \Rightarrow \mathtt{T}[p] \leq \mathtt{T}[q] \,\}$

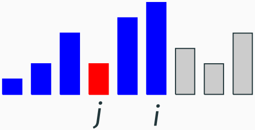
$\quad\quad$ swap$(T, j, j-1);$

$\quad\quad j := j - 1$

$\quad$ done

$\quad i := i + 1$

done



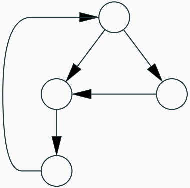
Plus an invariant: $\mathtt{T}$ is a permutation of the initial array $\mathtt{T}_0$

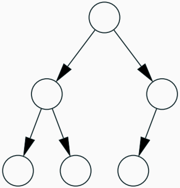
Pointers
and the Burstall-Bornat model

Pointers

Pointers: explicit (Algol-W, Pascal, C, C++) or implicit via objects passed by reference (Java, Lisp, Python, OCaml, ...).

Used to represent and operate on graphs and linked data structures (lists, trees, ...).





Example: singly-linked lists

```
class List {
  int head;
  List tail;
}
```

```
typedef struct cell * list;
struct cell { int head; list tail; };
```

The lists [1; 2; 3] and [4; 5] :

In-place concatenation of lists `l1` and `l2`:

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Example: singly-linked lists

```
class List {
  int head;
  List tail;
}
```

```
typedef struct cell * list;
struct cell { int head; list tail; };
```

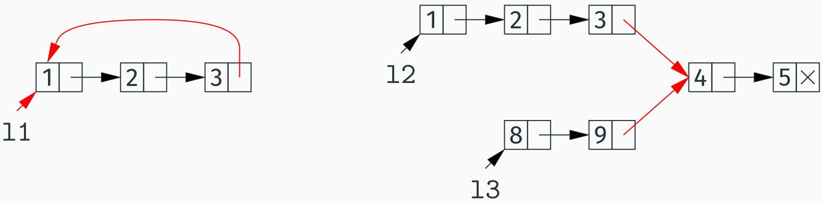
The lists [1; 2; 3] and [4; 5] :

In-place concatenation of lists `l1` and `l2`:

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Sharing memory cells

Sharing (having several pointers to the same cell) is essential to represent graphs, but problematic for simpler data structures.



Left:: `l1` is a cyclic (infinite) list $1, 2, 3, 1, 2, 3, \ldots$

Right: `l2` and `l3` share a suffix “4, 5”.

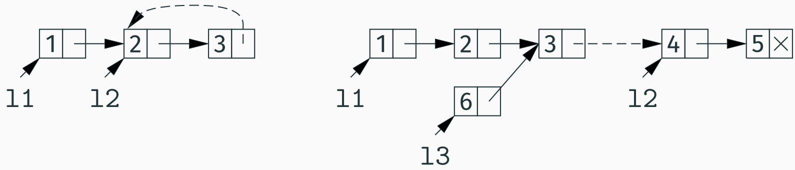
Problem with sharing

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

If `l1` is cyclic, concatenation does not terminate.

If `l1` and `l2` share cells, a cyclic list is created.

Any list `l3` that shares cells with `l1` is modified as a side effect.



Modeling pointers and the memory heap

Naive model:

- The memory heap = one big global array M.
- A pointer p = an index in M.
- An access $p \rightarrow \texttt{f}$ = an access $M[p + \texttt{offset(f)}]$.

The Burstall-Bornat model:

- The memory heap = one global array per field $F_1, F_2, \ldots$
- A pointer p = an index in the arrays F_i.
- An access $p \rightarrow \texttt{f}$ = an access $F[p]$.
- A write $p \rightarrow \texttt{f} := a$ modifies $F[p]$ but the other arrays $F' \neq F$ are unchanged.

Graphs in the Burstall-Bornat model

```
struct node {
    bool mark;
    int arity;
    struct node * child[arity];
}
```

Three global arrays $\texttt{MARK}[p]$, $\texttt{ARITY}[p]$, $\texttt{CHILD}[p][i]$.

The reachability relation $path(p, q)$,

"node q is reachable from node p".

$$path(p, p) \qquad \frac{p \neq 0 \qquad 0 \leq i < \texttt{ARITY}[p] \qquad path(\texttt{CHILD}[p][i], q)}{path(p, q)}$$

A generic algorithm for graph traversals

Mark all nodes reachable from a root r.

$$\{\, \forall x,\ \mathtt{MARK}[x] = 0 \,\}$$

$W := \{r\}$

$\mathtt{while}\ W \neq \emptyset\ \mathtt{do}$

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

  pick $p \in W$; $\quad W := W \setminus \{p\}$;

  $\mathtt{if}\ \mathtt{MARK}[p] = 0\ \mathtt{then}\ \mathtt{begin}$

    $\mathtt{MARK}[p] := 1$;

    $W := W \cup \{\mathtt{CHILD}[p][i] \mid 0 \leq i < \mathtt{ARITY}[p]\}$

  $\mathtt{end}$

$\mathtt{done}$

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

(Note: the write $\mathtt{MARK}[p] := 1$ leaves unchanged the arrays `CHILD` and `ARITY`, and therefore preserves the relation *path*.)

Singly-linked lists in the Burstall-Bornat model

Two global arrays `HEAD` and `TAIL`.

A representation predicate: $\mathit{lseg}(w, p, q)$,
"between pointers p and q lies the representation of the (mathematical) list w".

$$\mathit{lseg}(\varepsilon, p, p) \qquad \frac{p \neq 0 \qquad \mathtt{HEAD}[p] = n \qquad \mathit{lseg}(w, \mathtt{TAIL}[p], q)}{\mathit{lseg}(n \cdot w, p, q)}$$

p points to a well-formed list (without cycles) $=$
$$\exists w,\ \mathit{lseg}(w, p, \mathtt{NULL}).$$

p and q point to disjoint lists (no sharing) $=$
$$\forall r, w, w',\ \mathit{lseg}(w, p, r) \land \mathit{lseg}(w', q, r) \Rightarrow r = \mathtt{NULL}$$

A specification for list concatenation

Define $\mathit{list}(w, p) \stackrel{\mathit{def}}{=} \mathit{lseg}(w, p, \mathtt{NULL})$, “pointer p represents list w”.

$$\{\, w \neq \varepsilon \wedge \mathit{list}(w, \mathtt{l1}) \wedge \mathit{list}(w', \mathtt{l2}) \wedge \mathit{disjoint}(\mathtt{l1}, \mathtt{l2}) \,\}$$
$$\mathtt{concat}(\mathtt{l1}, \mathtt{l2})$$
$$\{\, \mathit{list}(w \cdot w', \mathtt{l1}) \wedge \mathit{list}(w', \mathtt{l2}) \,\}$$

A reasonable specification, but still incomplete: we miss the fact that any list `l3` initially disjoint from `l1` is not modified.

> *The verification proofs are quite long for such a simple matter and very boring. We will not weary the reader with them; instead we will try to do better.*
>
> *(R. M. Burstall, 1972)*

Local reasoning
and memory footprints

Local reasoning

A common-sense principle:

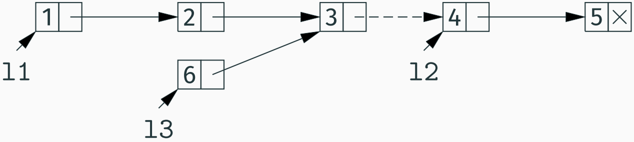
> *Everything that is not explicitly mentioned in* $\{\, P \,\}\; c \;\{\, Q \,\}$ *is preserved during the execution of* c*.*

In Hoare logic, this principle is expressed by the frame rule:

$$\frac{\{\, P \,\}\; c \;\{\, Q \,\} \qquad \text{no variable modified by } c \text{ appears in } R}{\{\, P \wedge R \,\}\; c \;\{\, Q \wedge R \,\}}$$

Example: $\{\, x = 0 \,\}\; x := x + 1 \;\{\, x = 1 \,\}$, therefore
$\{\, x = 0 \wedge y = 8 \,\}\; x := x + 1 \;\{\, x = 1 \wedge y = 8 \,\}$.

Pointers + sharing = no more local reasoning?



Consider

$$P = \mathit{list}(1.2.3.\varepsilon, \mathtt{l1}) \wedge \mathit{list}(4.5.\varepsilon, \mathtt{l2}) \wedge \mathit{disjoint}(\mathtt{l1}, \mathtt{l2})$$
$$Q = \mathit{list}(1.2.3.4.5.\varepsilon, \mathtt{l1})$$
$$R = \mathit{list}(6.3.\varepsilon, \mathtt{l3})$$

We do have $\{\, P \,\}\ \mathtt{concat}(\mathtt{l1}, \mathtt{l2})\ \{\, Q \,\}$

but not $\{\, P \wedge R \,\}\ \mathtt{concat}(\mathtt{l1}, \mathtt{l2})\ \{\, Q \wedge R \,\}$ (*R* is false “after”).

Towards a better frame rule

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad \text{no variable modified by } c \text{ appears in } R \qquad \text{no memory location modified by } c \text{ is mentioned in } R}{\{\,P \wedge R\,\}\ c\ \{\,Q \wedge R\,\}}$$

This rule is plausible, but the condition "no memory location modified by c is mentioned in R" is not syntactic.

It would be great if the program logic itself was able to verify this condition!

Memory footprints

To a logical assertion P, Q we associate a memory footprint: the set of memory locations (pointers) whose contents are described by the assertion.

Example

The assertion $p \mapsto 0$, “location p contains value 0”, has footprint $\{p\}$.

The assertion $p \mapsto 0 \land q \mapsto 1$ has footprint $\{p, q\}$.

The assertion $(b \land p \mapsto 0) \lor (\neg b \land q \mapsto 1)$ has footprint $\{p\}$ if b is true, $\{q\}$ if b is false.

Towards a better frame rule

Intuition (almost true): if $\{ P \} \; c \; \{ Q \}$, the memory locations modified during the execution of c are mentioned in P or in Q, and therefore belong to their footprints.

$$\frac{\begin{array}{c}\{ P \} \; c \; \{ Q \} \\ \text{no variable modified by } c \text{ appears in } R \\ \mathit{footprint}(P) \cap \mathit{footprint}(R) = \emptyset \\ \mathit{footprint}(Q) \cap \mathit{footprint}(R) = \emptyset\end{array}}{\{ P \wedge R \} \; c \; \{ Q \wedge R \}}$$

Separating conjunction

The statement “P and R are true and their footprints are disjoint” occurs so often that we give it a name: separating conjunction, written $P \star R$.

Formally: (assertions = predicates on the heap h)

$$(P \star R)\ h \stackrel{def}{=} \exists h_1, h_2,\ P\ h_1 \wedge R\ h_2 \wedge h = h_1 \uplus h_2 \text{ (disjoint union)}$$

Example

$p \mapsto 0 \star p \mapsto 0$ is always false.

$p \mapsto 0 \star q \mapsto 0$ implies $p \neq q$.

Separating conjunction and the frame rule

The frame rule from separation logic:

$$\frac{\{\, P \,\}\ c\ \{\, Q \,\} \qquad \text{no variable modified by } c \text{ appears in } R}{\{\, P \star R \,\}\ c\ \{\, Q \star R \,\}}$$

Captures elegantly the notion of local reasoning:

P, Q describe the parts of memory relevant for the execution of c;

R describes other parts of memory.

Separation logic

The path to separation logic

Burstall (1972): *Distinct Nonrepeating List Systems*
$\approx$ singly-linked structures without any sharing
+ ad-hoc reasoning rules.

Reynolds (1999), *Intuitionistic Reasoning about Shared Mutable Data Structures*. Introduces the notion of separating conjunction.

O'Hearn and Pym (1999), *The Logic of Bunched Implications*. Reasoning about resources that are used linearly.

O'Hearn, Reynolds, Yang (2001), *Local Reasoning about Programs that Alter Data Structures*. The modern presentation of separation logic.

Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*. The paper that gave separation logic its name.

Which language with pointers?

Classic approach: IMP + operations `alloc`, `get`, `set`, `free`.

Two degrees of mutability: variables and memory locations.

Assertions = predicates on the state of variables (the store s)
and on the memory state (the heap h)

In this lecture: mini-ML + references
in other words: lambda-calculus + state monad.

Immutable variables that contain references (pointers) to mutable memory locations.

Assertions = predicates on the memory state (the heap h).

The PTR language

Commands (expressions with effects):

$$
\begin{array}{rll}
c ::= & a & \text{pure expression} \\
\mid & \mathtt{let}\ x = c\ \mathtt{in}\ c' & \text{sequencing and binding} \\
\mid & \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 & \text{conditional} \\
\mid & \mathtt{choose}(N) & \text{nondeterministic choice} \\
\mid & \mathtt{alloc}(N) & \text{allocate } N \text{ memory locations} \\
\mid & \mathtt{get}(a) & \text{read from location } a \\
\mid & \mathtt{set}(a, a') & \text{write to location } a \\
\mid & \mathtt{free}(a) & \text{free location } a
\end{array}
$$

In examples, we will also use recursive functions $\mathtt{def}\ f\ x_1\ \cdots\ x_n = c$.

Examples of PTR programs

Allocation and initialization of a list cell:

```
def cons hd tl =
    let a = alloc(2) in
    let _ = set(a, hd) in
    let _ = set(a + 1, tl) in a
```

Note: locations are integers; pointer arithmetic is supported.

In-place concatenation of two lists:

```
def concat_rec l1 l2 =
    let tl = get(l1 + 1) in
    if tl = 0 then set(l1 + 1, l2) else concat_rec tl l2
def concat l1 l2 =
    if l1 = 0 then l2 else let _ = concat_rec l1 l2 in l1
```

Assertions

Assertions are predicates on the heap h.

Pure assertions (P is a proposition):

$$\langle P \rangle \stackrel{\mathit{def}}{=} \lambda h.\, P \wedge \mathit{Dom}(h) = \emptyset$$

"The memory is empty"

$$\mathit{emp} \stackrel{\mathit{def}}{=} \langle \top \rangle = \lambda h.\, \mathit{Dom}(h) = \emptyset$$

"Location ℓ contains value v"

$$\ell \mapsto v \stackrel{\mathit{def}}{=} \lambda h.\, \mathit{Dom}(h) = \{\ell\} \wedge h(\ell) = v$$

"Location ℓ is valid"

$$\ell \mapsto \_ \stackrel{\mathit{def}}{=} \exists v,\, \ell \mapsto v = \lambda h.\, \mathit{Dom}(h) = \{\ell\}$$

Separating conjunction

The separating conjunction $P \ast Q$ says that we can split the heap in two parts, one satisfying P and the other satisfying Q.

$$P \ast Q \stackrel{def}{=} \lambda h.\ \exists h_1, h_2,\ P\ h_1 \wedge Q\ h_2 \wedge h = h_1 \uplus h_2$$

Some properties:

$$P \ast Q = Q \ast P \qquad (P \ast Q) \ast R = P \ast (Q \ast R)$$

$$\mathtt{emp} \ast P = P \ast \mathtt{emp} = P \qquad \langle A \rangle \ast \langle B \rangle = \langle A \wedge B \rangle$$

The rules of separation logic

The rules define triples $\{ P \}\ c\ \{ Q \}$.

Since commands return values, the postcondition is a function $\lambda v \ldots$ from values to assertions.

$$\frac{P \Rightarrow Q\ [\![a]\!]}{\{ P \}\ a\ \{ Q \}} \qquad \frac{\forall n \in [0, N),\ P \Rightarrow Q\ n}{\{ P \}\ \mathtt{choose}(N)\ \{ Q \}}$$

$$\frac{\{ P \}\ c\ \{ R \} \qquad \forall v,\ \{ R\ v \}\ c'[x \leftarrow v]\ \{ Q \}}{\{ P \}\ \mathtt{let}\ x = c\ \mathtt{in}\ c'\ \{ Q \}}$$

$$\frac{\{ \langle b \rangle \star P \}\ c_1\ \{ Q \} \qquad \{ \langle \neg b \rangle \star P \}\ c_2\ \{ Q \}}{\{ P \}\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \{ Q \}}$$

Structural rules

$$\frac{\{\, P \,\}\; c\; \{\, Q \,\}}{\{\, P \ast R \,\}\; c\; \{\, \lambda v.\; Q\, v \ast R \,\}}\ \text{(frame)}$$

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\; c\; \{\, Q' \,\} \qquad \forall v,\; Q'\, v \Rightarrow Q\, v}{\{\, P \,\}\; c\; \{\, Q \,\}}\ \text{(consequence)}$$

$$\frac{A \Rightarrow \{\, P \,\}\; c\; \{\, Q \,\}}{\{\, \langle A \rangle \ast P \,\}\; c\; \{\, Q \,\}}\ \text{(pure-elim)} \qquad \frac{\forall x.\; \{\, P \,\}\; c\; \{\, Q \,\}}{\{\, \exists x. P \,\}\; c\; \{\, Q \,\}}\ \text{(}\exists\text{-elim)}$$

The “small rules” for heap operations

“Small” means “with the smallest footprint”.

$$\{\,\mathrm{emp}\,\}\quad \texttt{alloc}(N)\quad \{\,\lambda\ell.\ \ell \mapsto \_ \star \cdots \star \ell + N - 1 \mapsto \_\,\}$$

$$\{\,[\![a]\!] \mapsto x\,\}\quad \texttt{get}(a)\quad \{\,\lambda v.\ \langle v = x\rangle \star [\![a]\!] \mapsto x\,\}$$

$$\{\,[\![a]\!] \mapsto \_\,\}\quad \texttt{set}(a, a')\quad \{\,\lambda v.\ [\![a]\!] \mapsto [\![a']\!]\,\}$$

$$\{\,[\![a]\!] \mapsto \_\,\}\quad \texttt{free}(a)\quad \{\,\lambda v.\ \mathrm{emp}\,\}$$

“Large” rules are obtained by framing, e.g.

$$\{\,P\,\}\ \texttt{alloc}(2)\ \{\,\lambda\ell.\ P \star \ell \mapsto \_ \star \ell + 1 \mapsto \_\,\}$$

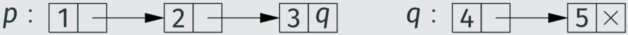
Data structures and representation predicates

Singly-linked lists

Representation predicates:

$$
\begin{aligned}
lseg(\varepsilon, p, q) &= \langle p = q \rangle \\
lseg(x \cdot w, p, q) &= \exists p', p \mapsto x \star p + 1 \mapsto p' \star lseg(w, p', q) \\
list(w, p) &= lseg(w, p, \texttt{NULL})
\end{aligned}
$$

Example



We have $lseg(1.2.3.\varepsilon, p, q) \star list(4.5.\varepsilon, q)$ or, equivalently, $list(1.2.3.4.5.\varepsilon, p)$.

Specifying functions on singly-liked lists

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{length}(p) \quad \{\, \lambda r.\, \langle r = |w| \rangle \ast \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{copy}(p) \quad \{\, \lambda r.\, \mathit{list}(w,r) \ast \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{dispose}(p) \quad \{\, \lambda r.\, \mathit{emp} \,\}$$

$$\{\, \mathit{list}(w,p) \ast \mathit{list}(w',q) \,\} \quad \texttt{concat}(p,q) \quad \{\, \lambda r.\, \mathit{list}(w \cdot w', r) \,\}$$

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{reverse}(p) \quad \{\, \lambda r.\, \mathit{list}(\mathit{rev}(w), r) \,\}$$

Specifying functions on singly-liked lists

$$\{\, list(w,p) \,\}\quad \texttt{length}(p) \quad \{\, \lambda r.\ \langle r = |w| \rangle \ast list(w,p) \,\}$$

$$\{\, list(w,p) \,\}\quad \texttt{copy}(p) \quad \{\, \lambda r.\ list(w,r) \ast list(w,p) \,\}$$

$$\{\, list(w,p) \,\}\quad \texttt{dispose}(p) \quad \{\, \lambda r.\ emp \,\}$$

$$\{\, list(w,p) \ast list(w',q) \,\}\quad \texttt{concat}(p,q) \quad \{\, \lambda r.\ list(w \cdot w', r) \,\}$$

$$\{\, list(w,p) \,\}\quad \texttt{reverse}(p) \quad \{\, \lambda r.\ list(rev(w), r) \,\}$$

Control of sharing:

- For $\texttt{copy}(p)$, the postcondition $list(w,r) \ast list(w,p)$ guarantees that the result and the argument are disjoint.
- For $\texttt{concat}(p,q)$, the precondition $list(w,p) \ast list(w',q)$ requires that the two arguments are disjoint.

Specifying functions on singly-liked lists

$$\{\, list(w,p) \,\} \quad \texttt{length}(p) \quad \{\, \lambda r.\, \langle r = |w| \rangle \ast list(w,p) \,\}$$

$$\{\, list(w,p) \,\} \quad \texttt{copy}(p) \quad \{\, \lambda r.\, list(w,r) \ast list(w,p) \,\}$$

$$\{\, list(w,p) \,\} \quad \texttt{dispose}(p) \quad \{\, \lambda r.\, emp \,\}$$

$$\{\, list(w,p) \ast list(w',q) \,\} \quad \texttt{concat}(p,q) \quad \{\, \lambda r.\, list(w \cdot w', r) \,\}$$

$$\{\, list(w,p) \,\} \quad \texttt{reverse}(p) \quad \{\, \lambda r.\, list(rev(w), r) \,\}$$

Resource management:

- Some lists are preserved (`length`, `copy`)
- Some lists are allocated (`copy`) or destroyed (`dispose`)
- Some lists are recycled into new lists (`concat`)

Specifying functions on singly-liked lists

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{length}(p) \quad \{\, \lambda r.\ \langle r = |w| \rangle \star \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{copy}(p) \quad \{\, \lambda r.\ \mathit{list}(w,r) \star \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{dispose}(p) \quad \{\, \lambda r.\ \mathit{emp} \,\}$$

$$\{\, \mathit{list}(w,p) \star \mathit{list}(w',q) \,\} \quad \texttt{concat}(p,q) \quad \{\, \lambda r.\ \mathit{list}(w \cdot w', r) \,\}$$

$$\{\, \mathit{list}(w,p) \,\} \quad \texttt{reverse}(p) \quad \{\, \lambda r.\ \mathit{list}(\mathit{rev}(w), r) \,\}$$

Permissions:

- After $\texttt{dispose}(p)$ or $\texttt{concat}(p,q)$, we lose the right to access p and q as well-formed lists.
- After $\texttt{concat}(p,q)$, we gain the right to access the result value as a well-formed list.

An example of verification

$\{\, list(w,p) \star list(w',q) \,\}$

```
def rev_append p q =
  if p = NULL then          // w = ε

    q
  else                      // w = x · w₁ for some x and w₁

    let t = get(p + 1) in

    let _ = set(p + 1, q) in

    rev_append t p
```

$\{\, \lambda r.\ list(rev(w) \cdot w', r) \,\}$

An example of verification

$\{\, list(w, p) \star list(w', q) \,\}$

def rev_append p q $=$

  if $p = \mathtt{NULL}$ then $// \; w = \varepsilon$

    $\{\, \langle p = \mathtt{NULL} \rangle \star list(w', q) \,\}$

   q

  else $// \; w = x \cdot w_1$ for some x and w_1

    $\{\, \exists p', p \mapsto x \star p + 1 \mapsto p' \star list(w_1, p') \star list(w', q) \,\}$

   let $t = \mathtt{get}(p + 1)$ in

    $\{\, p \mapsto x \star p + 1 \mapsto t \star list(w_1, t) \star list(w', q) \,\}$

   let _ $= \mathtt{set}(p + 1, q)$ in

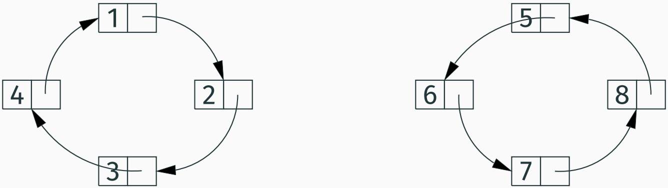
    $\{\, list(w_1, t) \star p \mapsto x \star p + 1 \mapsto q \star list(w', q) \,\}$

   rev_append t p

    $\{\, \lambda r.\, list(rev(w_1) \cdot x \cdot w', r) \,\}$

$\{\, \lambda r.\, list(rev(w) \cdot w', r) \,\}$

Circular lists



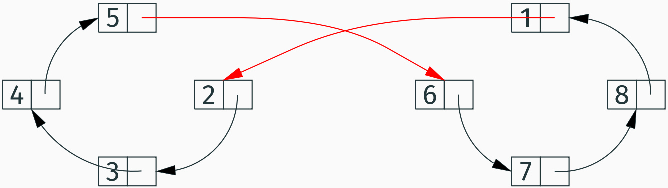
Representation predicates:

$$circlist(w, p) = \langle w \neq \varepsilon \rangle \star lseg(w, p, p)$$

In-place concatenation:

$$\begin{array}{c} \{\, circlist(w, p) \star circlist(w', q) \,\} \\ \mathtt{swap}(p, q); \mathtt{swap}(p+1, q+1) \\ \{\, circlist(w \cdot w', q) \,\} \end{array}$$

Circular lists



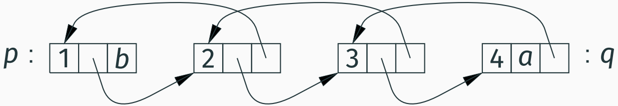
Representation predicates:

$$circlist(w, p) = \langle w \neq \varepsilon \rangle \star lseg(w, p, p)$$

In-place concatenation:

$$\begin{array}{c} \{\, circlist(w, p) \star circlist(w', q) \,\} \\ \mathtt{swap}(p, q); \mathtt{swap}(p+1, q+1) \\ \{\, circlist(w \cdot w', q) \,\} \end{array}$$

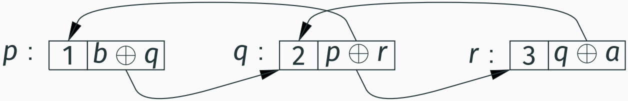
Doubly-linked lists



Forward chaining from *p* to *a* + backward chaining from *q* to *b*.

$$\begin{aligned} dlseg(\varepsilon, p, a, q, b) &= \langle p = a \wedge q = b \rangle \\ dlseg(x \cdot w, p, a, q, b) &= \exists p', p \mapsto x \star p + 1 \mapsto p' \star p + 2 \mapsto b \\ &\qquad \star\ dlseg(w, p', a, q, p') \\ dlist(w, p, q) &= dlseg(w, p, \mathtt{NULL}, q, \mathtt{NULL}) \end{aligned}$$

The “xor” trick



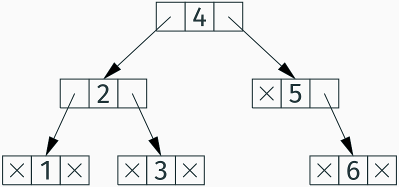
In each cell we store the “exclusive or” of the forward pointer and the backward pointer.

Forward traversal: we have p and q, we recover $r = (p \oplus r) \oplus p$.

Backward traversal: we have r and q, we recover $p = (p \oplus r) \oplus r$.

$$\begin{aligned}
dlseg(\varepsilon, p, a, q, b) &= \langle p = a \wedge q = b \rangle \\
dlseg(x \cdot w, p, a, q, b) &= \exists p', p \mapsto x \ast p + 1 \mapsto b \oplus p' \ast \\
&\qquad \ast\, dlseg(w, p', a, q, p') \\
dlist(w, p, q) &= dlseg(w, p, \mathtt{NULL}, q, \mathtt{NULL})
\end{aligned}$$

Binary trees



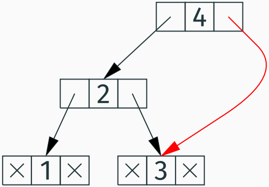
Representation predicate:

$$tree(\mathtt{Leaf}, p) = \langle p = \mathtt{NULL} \rangle$$

$$tree(\mathtt{Node}(t_1, x, t_2), p) = \exists p_1, p_2,\ p \mapsto p_1 \star p + 1 \mapsto x \star p + 2 \mapsto p_2 \star tree(t_1, p_1) \star tree(t_2, p_2)$$

Note: no internal sharing is allowed, the subtrees must be disjoint.

Binary trees with internal sharing ($\approx$ dags)



Possible if we use an *overlapping conjunction*: (Hobor and Villard, 2013)

$$(P \circledast Q)\ h = \exists h_1, h_2, h_3, h = h_1 \uplus h_2 \uplus h_3 \wedge P\ (h_1 \uplus h_2) \wedge Q\ (h_1 \uplus h_3)$$

$$\begin{aligned} \mathit{tree}(\texttt{Leaf}, p) &= \langle p = \texttt{NULL} \rangle \\ \mathit{tree}(\texttt{Node}(t_1, x, t_2), p) &= \exists p_1, p_2,\ p \mapsto p_1 \star p+1 \mapsto x \star p+2 \mapsto p_2 \\ &\qquad \star\ (\mathit{tree}(t_1, p_1) \circledast \mathit{tree}(t_2, p_2)) \end{aligned}$$

Semantic soundness of separation logic

Semantic soundness of separation logic

We gave rules that define triples $\{ P \}\ c\ \{ Q \}$.

If $\{ P \}\ c\ \{ Q \}$ can be derived by these rules, does all possible executions of c respect the contract stated by this triple?

Reduction semantics for PTR

We reduce configurations c/h where h : locations $\stackrel{fin}{\to}$ values is the current heap.

The rules for the pure constructs:

$$
\begin{aligned}
(\texttt{let}\ x = a\ \texttt{in}\ c)/h &\to c[x \leftarrow [\![a]\!]]/h \\
(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/h &\to (\texttt{let}\ x = c_1'\ \texttt{in}\ c_2)/h' \quad \text{if } c_1/h \to c_1'/h' \\
(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/h &\to \texttt{err} \quad \text{if } c_1/h \to \texttt{err} \\
(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/h &\to c_1/h \quad \text{if } [\![b]\!] \text{ is true} \\
(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/h &\to c_2/h \quad \text{if } [\![b]\!] \text{ is false} \\
\texttt{choose}(N)/h &\to n/h \quad \text{for any } n \in [0, N)
\end{aligned}
$$

Reduction semantics for PTR

The rules for the imperative constructs:

$$\texttt{alloc}(N)/h \to \ell/h[\ell \leftarrow 0, \ell+1 \leftarrow 0, \ldots, \ell+N-1 \leftarrow 0]$$
$$\text{for any } \ell \text{ such that } \{\ell, \ldots, \ell+N-1\} \cap Dom(h) = \emptyset$$
$$\texttt{get}(a)/h \to h([\![a]\!])/h \quad \text{if } [\![a]\!] \in Dom(h)$$
$$\texttt{set}(a, a')/h \to 0/h[\,[\![a]\!] \leftarrow [\![a']\!]\,] \quad \text{if } [\![a]\!] \in Dom(h)$$
$$\texttt{free}(a)/h \to 0/(h \setminus [\![a]\!]) \quad \text{if } [\![a]\!] \in Dom(h)$$

Error rules:

$$\texttt{get}(a)/h \to \texttt{err} \quad \text{if } [\![a]\!] \notin Dom(h)$$
$$\texttt{set}(a, a')/h \to \texttt{err} \quad \text{if } [\![a]\!] \notin Dom(h)$$
$$\texttt{free}(a)/h \to \texttt{err} \quad \text{if } [\![a]\!] \notin Dom(h)$$

Statement of semantic soundness

Same approach as for strong Hoare logic.

We define the inductive predicate $\texttt{Term}\ c\ h\ Q$, "command c started in state h always terminates without errors, in a state that satisfies Q".

$$\frac{Q\ [\![a]\!]\ h}{\texttt{Term}\ a\ h\ Q}$$

$$\frac{(\forall a,\ c \neq a) \qquad c/h \not\to \texttt{err} \qquad (\forall c', h',\ c/h \to c'/h' \Rightarrow \texttt{Term}\ c'\ h'\ Q)}{\texttt{Term}\ c\ h\ Q}$$

Semantic soundness

The semantic tripe: "if the initial state satisfies P, command c terminates in a state that satisfies Q"

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} \quad \stackrel{def}{=} \quad \forall h,\ P\ h \Rightarrow \texttt{Term}\ c\ h\ Q$$

We show that this definition validates the axioms and the inference rules of separation logic:

- If $P \Rightarrow Q\ [\![a]\!]$ then $\{\{\, P \,\}\}\ a\ \{\{\, Q \,\}\}$
- $\{\{\, [\![a]\!] \mapsto \_ \,\}\}\ \texttt{set}(a, a')\ \{\{\, \lambda v.\ [\![a]\!] \mapsto [\![a']\!] \,\}\}$
- etc.

Theorem (Semantic soundness of separation logic)

If $\{\, P \,\}\ c\ \{\, Q \,\}$ is derivable, then $\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\}$ holds.

Soundness of the frame rule

The main difficulty is to show that the frame rule is semantically valid:

$$\text{If } \{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} \text{ then } \{\{\, P \ast R \,\}\}\ c\ \{\{\, \lambda v.\ Q\ v \ast R \,\}\}.$$

To this end, we need a frame lemma for the `Term` predicate:

$$\text{If } \mathtt{Term}\ c\ h_1\ Q \text{ and } R\ h_2 \text{ then } \mathtt{Term}\ c\ (h_1 \uplus h_2)\ (\lambda v.\ Q\ v \ast R).$$

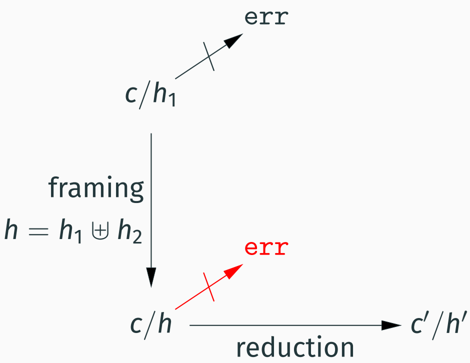
This holds because of a nice property of the operational semantics: if a command runs without errors in a "small" heap, every reduction step in a larger heap is simulated by a reduction step in the small heap.

Framing reductions

$$
\begin{tikzcd}
 & \texttt{err} \\
c/h_1 \arrow[ur, "/" marking] \arrow[d, "\begin{array}{r}\text{framing}\\ h = h_1 \uplus h_2\end{array}"'] & \\
c/h \arrow[r, "\text{reduction}"'] & c'/h'
\end{tikzcd}
$$

If $c/h_1 \not\to \texttt{err}$, then $c/h_1 \uplus h_2 \not\to \texttt{err}$. If, moreover, $c/h_1 \uplus h_2 \to c'/h'$, there exists h_1' such that $h' = h_1' \uplus h_2$ and $c/h_1 \to c'/h_1'$.

Framing reductions



If $c/h_1 \not\to \texttt{err}$, then $c/h_1 \uplus h_2 \not\to \texttt{err}$. If, moreover, $c/h_1 \uplus h_2 \to c'/h'$, there exists h'_1 such that $h' = h'_1 \uplus h_2$ and $c/h_1 \to c'/h'_1$.

Framing reductions

$$\begin{tikzcd}
& \mathtt{err} & \\
c/h_1 \arrow[ur, "|" marking] \arrow[rr, "\text{reduction}"'] \arrow[dd, "\begin{array}{r}\text{framing}\\ h = h_1 \uplus h_2\end{array}"'] & & c'/h'_1 \arrow[dd, "\begin{array}{l}\text{framing}\\ h' = h'_1 \uplus h_2\end{array}"] \\
& \mathtt{err} & \\
c/h \arrow[ur, "|" marking] \arrow[rr, "\text{reduction}"'] & & c'/h'
\end{tikzcd}$$

If $c/h_1 \not\to \mathtt{err}$, then $c/h_1 \uplus h_2 \not\to \mathtt{err}$. If, moreover, $c/h_1 \uplus h_2 \to c'/h'$, there exists h'_1 such that $h' = h'_1 \uplus h_2$ and $c/h_1 \to c'/h'_1$.

Framing reductions

This property holds in our PTR language because the reduction rule for allocations is nondeterministic: the allocated location ℓ can be chosen among all free locations.

$$\texttt{alloc}(N)/h \to \ell/h[\ell \leftarrow 0, \ell+1 \leftarrow 0, \ldots, \ell+N-1 \leftarrow 0]$$
$$\text{for any } \ell \text{ such that } \{\ell, \ldots, \ell+N-1\} \cap \mathit{Dom}(h) = \emptyset$$

This would not be the case if ℓ was a function of the heap:

$$\texttt{alloc}(N)/h \to \ell/h[\ell \leftarrow 0, \ell+1 \leftarrow 0, \ldots, \ell+N-1 \leftarrow 0]$$
$$\text{with } \ell = \texttt{firstfree}(h, N)$$

because, in general, $\texttt{firstfree}(h_1 \uplus h_2, N) \neq \texttt{firstfree}(h_1, N)$.

Plan B: validate the frame rule by construction

If allocation is deterministic, or if we would rather not prove the frame property for reductions, here is an alternative.

1. Define the usual semantic Hoare triple:

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\}_{Hoare} \ \stackrel{def}{=}\ \forall h,\ P\ h \Rightarrow \texttt{Term}\ c\ h\ Q$$

2. Define the semantic separation triple by quantifying over all possible framings:

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\}_{Sep} \ \stackrel{def}{=}\ \forall R,\ \{\{\, P \star R \,\}\}\ c\ \{\{\, \lambda v.\ Q\ v \star R \,\}\}_{Hoare}$$

Properties of the semantic Hoare triple

3. Show that the semantic Hoare triple $\{\!\{\, P \,\}\!\}\ c\ \{\!\{\, Q \,\}\!\}_{Hoare}$ validates

- the "large rules" for the imperative constructs

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

- the usual rules for the control structures:
    - if $P \Rightarrow Q\ [\![a]\!]$ then $\{\!\{\, P \,\}\!\}\ a\ \{\!\{\, Q \,\}\!\}_{Hoare}$
    - if $\{\!\{\, P \,\}\!\}\ c\ \{\!\{\, R \,\}\!\}_{Hoare}$ and $\forall v, \{\!\{\, Rv \,\}\!\}\ c'[x \leftarrow v]\ \{\!\{\, Q \,\}\!\}_{Hoare}$ then $\{\!\{\, P \,\}\!\}\ \texttt{let}\ x = c\ \texttt{in}\ c'\ \{\!\{\, Q \,\}\!\}_{Hoare}$
    - etc.
- but not the frame rule.

Properties of the semantic separation triple

$$\{\{\, P \,\}\}\; c\; \{\{\, Q \,\}\}_{Sep} \;\stackrel{def}{=}\; \forall R,\; \{\{\, P \star R \,\}\}\; c\; \{\{\, \lambda v.\, Q\, v \star R \,\}\}_{Hoare}$$

4. Notice that the semantic separation triple validates
   - the "small rules" for the imperative constructs;
   - the usual rules for the control structures;
   - the frame rule.
5. Conclude that $\{\, P \,\}\; c\; \{\, Q \,\}$ entails $\{\{\, P \,\}\}\; c\; \{\{\, Q \,\}\}_{Sep}$ and therefore $\{\{\, P \,\}\}\; c\; \{\{\, Q \,\}\}_{Hoare}$.

Summary

Summary so far

The emergence of separation logics in the early 2000's renewed the field of program logics and deductive verification entirely.

A great many extensions, ($\rightarrow$ lecture #5)
especially towards concurrency ($\rightarrow$ lectures #4 and #6)

Various implementations:

- deductive verification + automated theorem proving (Smallfoot, Infer, VeriFast) ($\rightarrow$ seminar #3)
- embeddings inside proof assistants (CFML, VST, Bedrock, IRIS) ($\rightarrow$ seminars #4 and #5)
- type systems such as that of the Rust language.

References

References

An overview of separation logic:

- Peter O'Hearn, *Separation Logic*, Comm. ACM 62(2), 2019.

One of the seminal papers, still a great reference today:

- John C. Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*, LICS 2002.

Mechanizing separation logic:

- The companion Coq development for this lecture `https://github.com/xavierleroy/cdf-program-logics`
- A. Charguéraud, *Foundations of Separation Logic*, 2021, `https://www.chargueraud.org/teach/verif/`