Mechanized semantics, fourth lecture

Logics to reason about programs

Xavier Leroy

2020-01-09

Collège de France, chair of software sciences

Reasoning about a program

Verify (with mathematical rigor) that a program or program fragment “behaves correctly”:

- Full correctness: the program terminates and produces the expected result.
- Partial correctness: if the program terminates, it produces the expected result.
- Robustness: the program does not crash (no run-time errors), does not leak confidential information, etc.

Reasoning about a program

In principle: it suffices to have a formal semantics for the programming language we use; then, we reason directly about the possible executions of the program of interest.

(See examples in the Coq file `HoareLogic.v`)

Reasoning about a program

In principle: it suffices to have a formal semantics for the programming language we use; then, we reason directly about the possible executions of the program of interest.

(See examples in the Coq file `HoareLogic.v`)

In practice: it is much more convenient to use higher-level reasoning principles, namely a program logic.

An idea as old as computing

Alan Turing, *Checking a large routine*, **1949**.

> Friday, 24th June.
>
> Checking a large routine. by Dr. A. Turing.
>
> How can one check a routine in the sense of making sure that it is right?
>
> In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Talk given at the *inaugural conference of the EDSAC computer*, Cambridge University, june 1949. The manuscript was corrected, commented and republished by F.L. Morris and C.B. Jones in *Annals of the History of Computing*, 6, 1984.

Turing's "large routine"

Compute the factorial function $n!$ using only additions.

Two nested loops.

```
int fac (int n)
{
  int s, r, u, v;
  u = 1;
  for (r = 1; r < n; r++) {
    v = u; s = 1;
    do {
      u = u + v;
    } while (s++ < r);
  }
  return u;
}
```

Turing's "large routine"

No structured programming in 1949: just flowcharts.

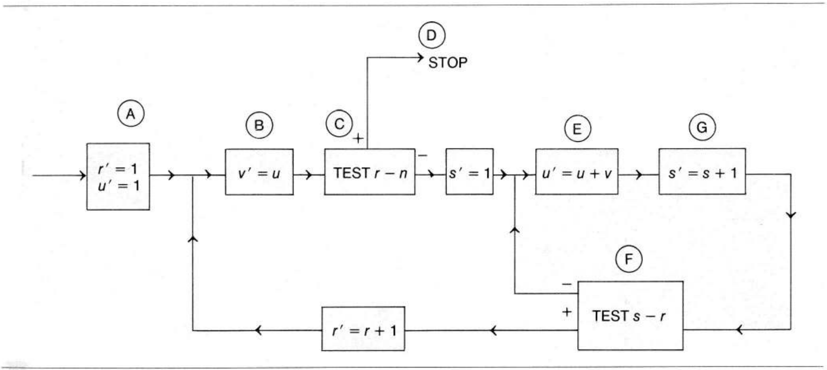


Figure 1 (Redrawn from Turing's original)

Turing's genius idea

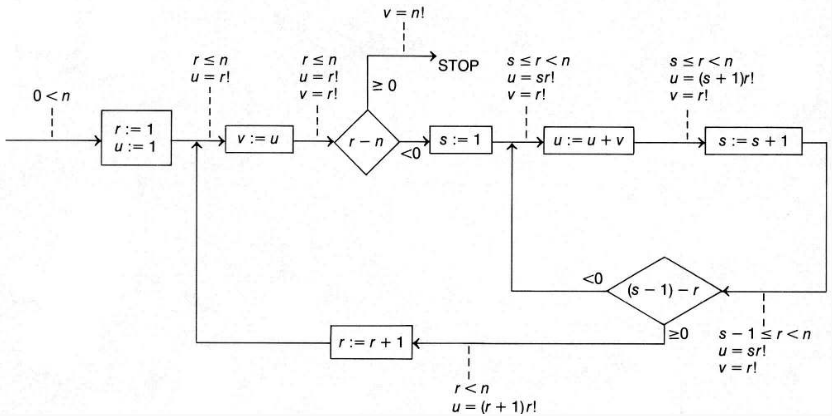
To each program point, associate a logical invariant:
a relation between the values of the variables that holds at every execution.

STORAGE LOCATION	(INITIAL) A $k = 6$	B $k = 5$	C $k = 4$	(STOP) D $k = 0$	E $k = 3$	F $k = 1$	G $k = 2$
27 28 29 30 31	 n 	 r n $\lfloor r$ 	 r n $\lfloor r$ $\lfloor r$	 n $\lfloor n$	s r n $s\lfloor r$ $\lfloor r$	$s + 1$ r n $(s + 1)\lfloor r$ $\lfloor r$	s r n $(s + 1)\lfloor r$ $\lfloor r$
	TO B WITH $r' = 1$ $u' = 1$	TO C	TO D IF $r = n$ TO E IF $r < n$		TO G	TO B WITH $r' = r + 1$ IF $s \geq r$ TO E WITH $s' = s + 1$ IF $s < r$	TO F

Figure 2 (Redrawn from Turing's original)

Turing's genius idea

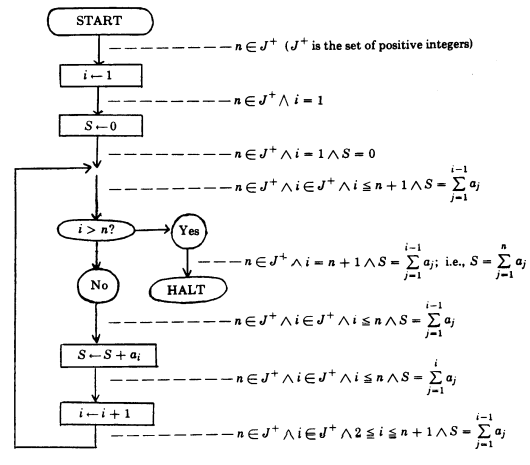
In more modern notation:



To verify the program, it suffices to check that every assertion logically implies the assertions at successor points.

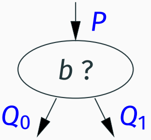
Robert Floyd, *Assigning meanings to programs*, 1967

18 years later, Floyd rediscovers and generalizes Turing's idea.

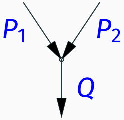


Robert Floyd, *Assigning meanings to programs*, 1967

Formalizes the logical rules that connect the preconditions P and the postconditions Q of flowchart nodes.



$$P \wedge \neg b \Rightarrow Q_0$$
$$P \wedge b \Rightarrow Q_1$$



$$P_1 \vee P_2 \Rightarrow Q$$

Floyd's rule for assignment

Examples:

$\{y \leq 10\}$ $x := 0$ $\{x = 0 \land y \leq 10\}$

$\{y = 2x\}$ $x := 0$ $\{x = 0 \land \exists x_0, y = 2x_0\}$

$\{0 \leq x \leq y\}$ $x := x + 1$ $\{\exists x_0, x = x_0 + 1 \land 0 \leq x_0 \leq y\}$

The general case:

$\{P(x, \vec{y})\}$ $x := f(x, \vec{y})$ $\{Q\}$

$$(\exists x_0, x = f(x_0, \vec{y}) \land P(x_0, \vec{y})) \Rightarrow Q$$

Robert Floyd, *Assigning meanings to programs*, 1967

Formalizes the logical rules to annotate a flowchart.

Observes that these rules define a semantics for the language.
(The birth of axiomatic semantics.)

Proves that these rules are sound with respect to an intuitive operational semantics.

Proves that these rules are complete.

Outlines extra conditions to guarantee termination.

Outlines an extension to the Algol 60 language (structured loops).

C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969

Reformulates Floyd's approach for structured control
(`if`/`then`/`else`, loops, ...) instead of flowcharts.

Presented by axioms and inference rules
$\rightarrow$ a logic to reason about programs
(just like Euclidean geometry is a logic to reason about figures).

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

(Communications of the ACM, 12(10), 1969)

Hoare logics

Hoare triples

The statements of Hoare logic:

$$[\,P\,]\; c\; [\,Q\,] \qquad\qquad \{\,P\,\}\; c\; \{\,Q\,\}$$

c: command in an imperative structured language (IMP, Algol, …)

P, Q: logical assertions about program variables.

P: precondition, assumed true “before” executing c

Q: postcondition, guaranteed true “after” executing c

Hoare triples

"Strong" Hoare logic: (total correctness)

$[\,P\,]\; c\; [\,Q\,]$ if P holds "before",
then c terminates and Q holds "after"

"Weak" Hoare logic: (partial correctness)

$\{\,P\,\}\; c\; \{\,Q\,\}$ if P holds "before" and if c terminates,
then Q holds "after"

The rules of weak Hoare logic

Structured control:

$$\frac{\{\,P\,\}\; c_1\; \{\,Q\,\} \qquad \{\,Q\,\}\; c_2\; \{\,R\,\}}{\{\,P\,\}\; c_1; c_2\; \{\,R\,\}}$$

$$\frac{\{\,P \wedge b\,\}\; c_1\; \{\,Q\,\} \qquad \{\,P \wedge \neg b\,\}\; c_2\; \{\,Q\,\}}{\{\,P\,\}\; \texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2\; \{\,Q\,\}}$$

$$\frac{\{\,P \wedge b\,\}\; c\; \{\,P\,\}}{\{\,P\,\}\; \texttt{while}\; b\; \texttt{do}\; c\; \{\,P \wedge \neg b\,\}}$$

The rules of weak Hoare logic

Empty command:

$$\{\, P \,\}\ \mathtt{SKIP}\ \{\, P \,\}$$

Assignment:

$$\{\, Q[x \leftarrow a] \,\}\ x := a\ \{\, Q \,\}$$

Note the “backward” style: the postcondition Q determines the precondition.

Example

$\{\, 0 = 0 \land y \leq 10 \,\}\ x := 0\ \{\, x = 0 \land y \leq 10 \,\}$

$\{\, 1 \leq x + 1 \leq 11 \,\}\ x := x + 1\ \{\, 1 \leq x \leq 11 \,\}$

The consequence rule

Enables us to replace preconditions and postconditions by equivalent or weaker formulas.

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\; c\; \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\; c\; \{\, Q \,\}}$$

Example

$$\frac{\begin{array}{c} 0 \leq x \leq 10 \Rightarrow 1 \leq x + 1 \leq 11 \\ \{\, 1 \leq x + 1 \leq 11 \,\}\; x := x + 1\; \{\, 1 \leq x \leq 11 \,\} \\ 1 \leq x \leq 11 \Rightarrow 1 \leq x \leq 11 \end{array}}{\{\, 0 \leq x \leq 10 \,\}\; x := x + 1\; \{\, 1 \leq x \leq 11 \,\}}$$

Strong Hoare logic

Same rules as for the weak logic, except for loops.

$$[\,P\,]\ \texttt{SKIP}\ [\,P\,] \qquad [\,Q[x \leftarrow a]\,]\ x := a\ [\,Q\,]$$

$$\frac{[\,P\,]\ c_1\ [\,Q\,] \quad [\,Q\,]\ c_2\ [\,R\,]}{[\,P\,]\ c_1; c_2\ [\,R\,]} \qquad \frac{[\,P \land b\,]\ c_1\ [\,Q\,] \quad [\,P \land \neg b\,]\ c_2\ [\,Q\,]}{[\,P\,]\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ [\,Q\,]}$$

$$\frac{P \Rightarrow P' \quad [\,P'\,]\ c\ [\,Q'\,] \quad Q' \Rightarrow Q}{[\,P\,]\ c\ [\,Q\,]}$$

Proving loop termination

No general rule, but one rule is often sufficient:
an expression V (the "variant") is a nonnegative integer and decreases strictly at every loop iteration.

$$\frac{\forall n \in \mathbb{Z},\ [\,P \land b \land V = n\,]\ c\ [\,P \land 0 \leq V < n\,]}{[\,P\,]\ \texttt{while}\ b\ \texttt{do}\ c\ [\,P \land \neg b\,]}$$

Mechanizing Hoare logic

How should we represent logical assertions?

By a dedicated language with its own syntax and semantics.

(deep embedding)

Terms: $t ::= x \mid 0 \mid 1 \mid t_1 + t_2 \mid \ldots$

Assertions: $P, Q ::= t_1 = t_2 \mid P \wedge Q \mid \forall x. P \mid \ldots$

By a predicate in Coq's logic. *(shallow embedding)*

$$P, Q : \texttt{store} \to \texttt{Prop}$$

Example: "$0 \leq x < y$" becomes `fun s -> 0 <= s "x" < s "y"`.

(See the Coq file `HoareLogic`, sections 4.2 and 4.3)

Syntactic triples, semantic triples

Syntactic approach: $\{ P \}\ c\ \{ Q \}$ can be derived from the axioms and inference rules of Hoare logic.

Semantic approach: $\{ P \}\ c\ \{ Q \}$ holds if and only iff $\forall s, s',\ P\ s \wedge c/s \Downarrow s' \Rightarrow Q\ s'$.

The two notions are equivalent!

- Soundness: if $\{ P \}\ c\ \{ Q \}$ can be derived by Hoare's rules, it is semantically true.
- Relative completeness: if $\{ P \}\ c\ \{ Q \}$ is semantically true, it can be derived from Hoare's rules.

(See the Coq file `HoareLogic`, sections 4.4 and 4.5)

Automating Hoare logic

In general, it is undecidable whether a Hoare triple holds.
(The triple $\{\, True \,\}\ c\ \{\, False \,\}$ holds iff c does not terminate.)

However, many deduction steps in Hoare logic are directed by the syntax of the command. For instance:

$$\{\, P \,\}\ x_1 := a_1; \ldots ; x_n := a_n\ \{\, Q \,\}$$

We can apply the assignment rule n times then the consequence rule on the left, obtaining the verification condition

$$P \Rightarrow Q[x_n \leftarrow a_n][\cdots][x_1 \leftarrow a_1]$$

This is a first-order logical formula that lends itself well to automated theorem proving.

Generating verification condition

Consider a command c where `while` loops are manually annotated with loop invariants Inv.

We can automatically produce a first-order logical formula $\texttt{vcgen}\ P\ c\ Q$ that is true if and only if the triple $\{\,P\,\}\ c\ \{\,Q\,\}$ holds in Hoare logic.

This is the approach followed by deductive verification tools such as ESC/Java, Frama-C, KeY, Why3, …

(See the Coq file `HoareLogics`, section 4.8.)

Extension to arrays

Incorrect: $\{\, Q\{t[a_1] \leftarrow a_2\} \,\}\; t[a_1] := a_2 \;\{\, Q \,\}$ ✘

(Not only $t[a_1]$ is modified, but also all $t[a]$ for every a that has the same value as a_1.)

Correct: $\{\, Q\{t \leftarrow t[a_1 \mapsto a_2]\} \,\}\; t[a_1] := a_2 \;\{\, Q \,\}$ ✔

The expression $t[a_1 \mapsto a_2]$ stands for an array identical to t except that index a_1 has value a_2.

We reason over these array expressions using the equation

$$(t[a_1 \mapsto a_2])\,[a] = \begin{cases} a_2 & \text{if } a = a_1 \\ t[a] & \text{if } a \neq a_1 \end{cases}$$

Extension to pointers and dynamic allocation

Example: singly-linked lists.


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

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

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

Extension to pointers and dynamic allocation

Example: singly-linked lists.


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

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

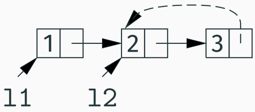
```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

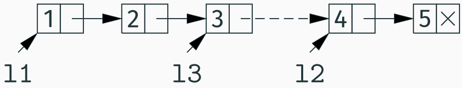
Pointers and linked data structures

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

Hard to verify this code... and even to specify it!

- List `l1` must be well formed (not circular) (otherwise the `while` loop does not terminate).
- List `l2` must not share any cell with `l1` (otherwise the concatenation builds a circular list).
- Any list `l3` that shares with `l1` is modified.





Hoare logics for pointers

Folklore approach: a pointer = an index in a big global array (the memory heap).

Burstall (1972), Morris (1981), Bornat (2000):
one heap per field of memory cells.

$$p \rightarrow \texttt{tail} := a \quad \stackrel{def}{=} \quad \texttt{tail} := \texttt{tail}[p \mapsto a] \qquad \text{(}\texttt{head}\text{ is unchanged)}$$

Bornat (2000), Mehta & Nipkow (2003), Hubert & Marché (2005):
mechanization of the “one heap per field” approach;
verifications of the Schorr-Waite graph traversal algorithm.

Separation logics

The path to separation logic

Burstall (1972): the *Distinct Nonrepeating List Systems*
($\approx$ simply-linked data 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*.
To reason 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.

Local reasoning

A common sense idea:

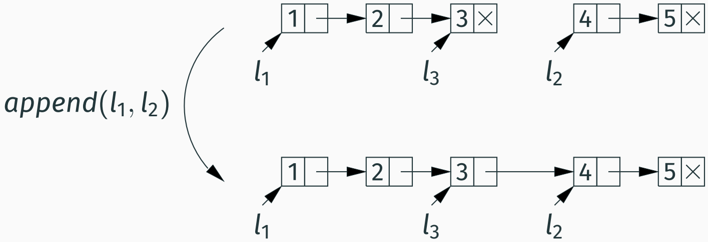
> *Anything that is not explicitly mentioned in* $\{P\}\ c\ \{Q\}$ *is preserved when executing* c*.*

In Hoare logic, this principle is presented as the following frame rule:

$$\frac{\{P\}\ c\ \{Q\} \qquad \text{none of the variables modified by } c \text{ appear in } R}{\{P \wedge R\}\ c\ \{Q \wedge R\}}$$

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

Pointers and aliasing ⇒ no more local reasoning



P = "l_1 represents list $[1, 2, 3]$ and l_2 represents list $[4, 5]$"
Q = "l_1 represents list $[1, 2, 3, 4, 5]$"
R = "l_3 represents list $[3]$".

$\{ P \}$ $append(l_1, l_2)$ $\{ Q \}$ is a valid triple, but not
$\{ P \wedge R \}$ $append(l_1, l_2)$ $\{ Q \wedge R \}$.

Memory footprint and separating conjunction

Every logical assertion P, Q has a memory footprint:
the set of memory locations (pointers) whose contents are described.

Example: assertion $p \mapsto 0$, “at location p there is value 0”, has the footprint $\{p\}$.

The separating conjunction $P * Q$ holds if and only if

- P holds (in the current memory state)
- Q holds (in the current memory state)
- P and Q have disjoint memory footprints.

Example: $p \mapsto 0 * p \mapsto 0$ is always false.
$p \mapsto 0 * q \mapsto 1$ implies $p \neq q$.

Representation predicates

Using separating conjunction, we can define the predicate $\mathit{list}(p, L)$, "pointer p is the head of a well-formed linked list that represents the abstract list L":

$$\begin{aligned} \mathit{list}(p, x :: L) &= \exists q,\ p \mapsto \{\texttt{head} = x; \texttt{tail} = q\} * \mathit{list}(q, L) \\ \mathit{list}(p, \mathit{nil}) &= p = \texttt{NULL} \end{aligned}$$

Separating conjunctions $\Rightarrow$ no internal sharing
(all list cells are pairwise distinct).

The memory footprint of $\texttt{list}(p, L)$ is the set of memory locations that are involved in the representation of L.

A specification in separation logic

In-place list concatenation:

for all abstract lists L, L', assuming L is not empty,

$$\{\ \mathit{list}(\texttt{l1}, L) * \mathit{list}(\texttt{l2}, L')\ \}$$

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

$$\{\ \mathit{list}(\texttt{l1}, L.L')\ \}$$

Separating conjunction in the precondition
$\Rightarrow$ no external sharing between `l1` and `l2`
(no list cell in common).

Nothing about `l2` in the postcondition
$\Rightarrow$ `l2` is no longer usable as a well-formed linked list.

Separating conjunction and the frame rule

The frame rule in separation logic:

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad \text{none of the variables modified by } c \text{ appear in } R}{\{\,P * R\,\}\ c\ \{\,Q * R\,\}}$$

Notion of local reasoning: P, Q describe the parts of memory relevant to the execution of c; R describes the other parts.

Notion of resources:

P describes the memory resources consumed by c;

Q describes the resources produced or returned by c;

R describes the resources untouched by c.

“Small rules”

Preconditions and postconditions only mention what is relevant to the execution of the command.

$$
\begin{array}{rcl}
[\,a = n\,] & x := a & [\,x = n\,] \\
[\,(a = p) * (p \mapsto v)\,] & x := *a & [\,(x = v) * (p \mapsto v)\,] \\
[\,(a = p) * (a' = v) * (p \mapsto \_)\,] & *a := a' & [\,p \mapsto v\,] \\
[\,emp\,] & x := alloc(N) & [\,\exists p, (x = p) \\
 & & \quad * (p \mapsto \_) * \cdots \\
 & & \quad * (p + N - 1 \mapsto \_)\,] \\
[\,(a = p) * (p \mapsto \_)\,] & free(a) & [\,emp\,]
\end{array}
$$

$p \mapsto \_$ reads as “p is valid” and is defined as $\exists v, p \mapsto v$.

Formalization: IMP with pointers

Commands:

$$
\begin{array}{llr}
c ::= \mathtt{SKIP} \mid x := a \mid c_1; c_2 & & \\
\quad \mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 & & \\
\quad \mid \mathtt{while}\ b\ \mathtt{do}\ c & & \\
\quad \mid x := \mathit{alloc}(N) & & \text{allocate } N \text{ words} \\
\quad \mid x := *a & & \text{read from location } a \\
\quad \mid *a_1 := a_2 & & \text{write to location } a_1 \\
\quad \mid \mathit{free}(a) & & \text{free location } a
\end{array}
$$

Pointers are integer values $\Rightarrow$ pointer arithmetic.

Example (Constructing a list)

```
l := alloc(2); *l := head; *(l + 1) := tail
```

Operational semantics

Two components for the memory state:

- the *store* s: variable $\mapsto$ value (total function)
- the *heap* h: location $\mapsto$ value (partial, finite function)

Reduction semantics: $c/s/h \rightarrow c'/s'/h'$.

Some representative rules:

$$x := a/s/h \rightarrow \texttt{SKIP}/s[x \leftarrow [\![a]\!]\ s]/h$$

$$x := *a/s/h \rightarrow \texttt{SKIP}/s[x \leftarrow v]/h \qquad \text{if } h([\![a]\!]\ s) = v$$

$$*a := a'/s/h \rightarrow \texttt{SKIP}/s/h[[\![a]\!]\ s \leftarrow [\![a']\!]\ s] \qquad \text{if } [\![a]\!]\ s \in dom(h)$$

A separation logic for IMP

Logical assertions = predicates $\texttt{store} \to \texttt{heap} \to \texttt{Prop}$.

Basic strong triples :

$$[\,P\,]\ c\ [\,Q\,] \stackrel{def}{=} \forall s, h,\ P\ s\ h \Rightarrow \exists s', h',\ c/s/h \stackrel{*}{\to} \texttt{SKIP}/s'/h' \wedge Q\ s'\ h'$$

Fail to validate the frame rule (because of dynamic allocation).

Strong triples:

$$[[\,P\,]]\ c\ [[\,Q\,]] \stackrel{def}{=} \forall R \text{ unchanged by } c, [\,P * R\,]\ c\ [\,Q * R\,]$$

Validate the frame rule.

(See Coq file `SepLogic` and A. Charguéraud’s seminar Jan. 16th.)

Extension: concurrent separation logic

(O'Hearn, 2007, *Resources, Concurrency and Local Reasoning*.
Brookes, 2007, *A Semantics for Concurrent Separation Logic*.)

Context: shared-memory concurrency
(threads, multicore processors, etc).

Base rule: parallel execution without interference.

$$\frac{\{\, P_1 \,\}\ c_1\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ c_2\ \{\, Q_2 \,\}}{\{\, P_1 * P_2 \,\}\ c_1 \parallel c_2\ \{\, Q_1 * Q_2 \,\}}$$

Synchronization and communication

Various rules to account for synchronization and communication between threads:

- High level: locks, semaphores, message queues, …
- Low level: memory barriers, compare-and-swap, load-acquire/store-release, …

A concurrent separation logic for locks

To a lock L we associate an assertion $INV(L)$:

- The footprint of $INV(L)$ describes the memory locations that are protected by the lock.
- The assertion $INV(L)$ describes the invariant that users of these protected locations must preserve.

Small rules for locks:

$$\{\, emp \,\} \quad \texttt{lock}(L) \quad \{\, INV(L) * Locked(L) \,\}$$
$$\{\, INV(L) * Locked(L) \,\} \quad \texttt{unlock}(L) \quad \{\, emp \,\}$$

Holding the lock = being the sole owner of protected locations.
Unlocking the lock = being obliged to restore the invariant.

Summary and perspectives

Summary

Two viewpoints that coexist nicely:

- Axiomatic viewpoint: a program logic defines the semantics of the programming language.
- Operational viewpoint: a program logic is a set of theorems about the operational semantics of the language, theorems which facilitate reasoning about programs.

Summary

The basic principles have been known for a long time,

and have remained purely theoretical for a long time,

but are now usable in practice thanks to tools:

- Deductive verifiers + automatic theorem provers:
  KeY, Frama-C WP, Infer, …
- Embeddings in Coq and other interactive theorem provers:
  CFML, IRIS, …

Perspectives

A very active research area:

- More automation for program proof.
  (E.g. INFER and “bi-abduction”; shape analyses.)
- More abstraction in program logics.
  (E.g. IRIS: monoïd + invariants = a concurrent logic)
- Reasoning on other kinds of resources.
  (E.g. file systems, execution time.)
- Reasoning on hardware-level concurrency.
  (Weakly-consistent memory models.)

References

References

Hoare logic and its mechanization:

- Benjamin Pierce et al, *Software Foundations, volume 2: Programming Languages Foundations*, chapters *Hoare logic*.
- Tobias Nipkow et Gerwin Klein, *Concrete Semantics*, chap. 12.

An introduction to separation logic:

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

Lectures on concurrent separation logics:

- Aleks Nanevski, *Separation Logic and Concurrency*, OPLSS 2016.
- Lars Birkedal, Ales Bizjak, *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*, 2018.