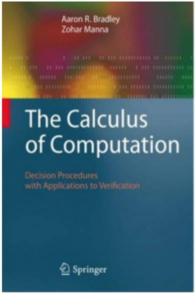
AAA616: Program Analysis

Lecture 10 — Logical Reasoning of Programs

Hakjoo Oh
2016 Fall

Reference

- The Calculus of Computation (Aaron Bradley and Zohar Manna)



Contents

- Propositional Logic (Chap 1)
- First-Order Logic (Chap 2, 3)
- Program Verification (Chap 5)

Motivating Example: Program-Equivalence Checking

Original Code	Optimized Code

`if (!a && !b) h();`
`else if (!a) g();`
`else f();`
	`if (a) f();`
	`else if (b) g ();`
	`else h();`

Motivating Example: Program-Equivalence Checking

Original Code	Optimized Code

`if (!a && !b) h();`
`else if (!a) g();`
`else f();`
	`if (a) f();`
	`else if (b) g ();`
	`else h();`

1. Treat procedures as independent boolean variables.
2. Translate if-then-else into boolean formula:

$$\text{if } x \text{ then } y \text{ else } z \equiv (x \land y) \lor (\neg x \land z)$$

3. Check equivalence of boolean formulas by a SAT Solver:

$$(\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f)$$
$$\iff a \land f \lor \neg a \land (b \land g \lor \neg b \land h)$$

Syntax of Propositional Logic

- An *atom* is a truth symbols $\bot, \top$ or propositional variables $P, Q, \ldots$.
- A *literal* is an atom α or its negation $\neg\alpha$.
- A *formula* is a literal or the application of a logical connectives:

$$
\begin{array}{rcl}
F & \rightarrow & \bot \\
& | & \top \\
& | & P \\
& | & \neg F \\
& | & F_1 \wedge F_2 \\
& | & F_1 \vee F_2 \\
& | & F_1 \rightarrow F_2 \\
& | & F_1 \leftrightarrow F_2
\end{array}
$$

Semantics of Propositional Logic

- An *interpretation* I assigns to every propositional variable exactly one truth value: e.g.,

$$I : \{P \mapsto \textbf{true}, Q \mapsto \textbf{false}, \ldots\}$$

- We write $I \models F$ if F evaluates to **true** under I.
- We write $I \not\models F$ if F evaluates to **false** under I.
- Semantics:

$$\begin{array}{lll}
I \models \top, \quad I \not\models \bot, & & \\
I \models P & \text{iff} & I[P] = \textbf{true} \\
I \not\models P & \text{iff} & I[P] = \textbf{false} \\
I \models \neg F & \text{iff} & I \not\models F \\
I \models F_1 \wedge F_2 & \text{iff} & I \models F_1 \text{ and } I \models F_2 \\
I \models F_1 \vee F_2 & \text{iff} & I \models F_1 \text{ or } I \models F_2 \\
I \models F_1 \rightarrow F_2 & \text{iff} & I \not\models F_1 \text{ or } I \models F_2 \\
I \models F_1 \leftrightarrow F_2 & \text{iff} & (I \models F_1 \text{ and } I \models F_2) \text{ or } (I \not\models F_1 \text{ and } I \not\models F_2)
\end{array}$$

Satisfiability and Validity

- A formula F is *satisfiable* iff there exists an interpretation I such that $I \models F$.
- A formula F is *valid* iff for all interpretations I, $I \models F$.
- Satisfiability and validity are dual concepts:

$$F \text{ is valid iff } \neg F \text{ is unsatisfiable.}$$

- We can check satisfiability by deciding validity, and vice versa.

Deciding Validity and Satisfiability

Two approaches to show F is valid:

- Truth table method performs exhaustive search: e.g.,

$$F : P \land Q \rightarrow P \lor \neg Q.$$

P	Q	$P \land Q$	$\neg Q$	$P \lor \neg Q$	F
0	0	0	1	1	1
0	1	0	0	0	1
1	0	0	1	1	1
1	1	1	0	1	1

- Semantic argument method uses deduction:
  - Assume F is invalid: $I \not\models F$ for some I.
  - Apply deduction rules to derive a contradiction.
  - If every branch of the proof derives a contradiction, then F is valid.
  - If some branch of the proof never derives a contradiction, then F is invalid.

Deduction Rules for Propositional Logic

$$\frac{I \models \neg F}{I \not\models F} \qquad \frac{I \not\models \neg F}{I \models F}$$

$$\frac{I \models F \wedge G}{I \models F, I \models G} \qquad \frac{I \not\models F \wedge G}{I \not\models F \mid I \not\models G}$$

$$\frac{I \models F \vee G}{I \models F \mid I \models G} \qquad \frac{I \not\models F \vee G}{I \not\models F, I \not\models G}$$

$$\frac{I \models F \rightarrow G}{I \not\models F \mid I \models G} \qquad \frac{I \not\models F \rightarrow G}{I \models F, I \not\models G}$$

$$\frac{I \models F \leftrightarrow G}{I \models F \wedge G \mid I \models \neg F \wedge \neg G} \qquad \frac{I \not\models F \leftrightarrow G}{I \models F \wedge \neg G \mid I \models \neg F \wedge G}$$

$$\frac{I \models F \quad I \not\models F}{I \models \bot}$$

Example 1

To prove that the formula

$$F : P \land Q \rightarrow P \lor \neg Q$$

is valid, assume that it is invalid and derive a contradiction:

1.	$I \not\models P \land Q \rightarrow P \lor \neg Q$	assumption
2.	$I \models P \land Q$	by 1 and semantics of $\rightarrow$
3.	$I \not\models P \lor \neg Q$	by 1 and semantics of $\rightarrow$
4.	$I \models P$	by 2 and semantics of $\land$
5.	$I \not\models P$	by 3 and semantics of $\lor$
6.	$I \models \bot$	4 and 5 are contradictory

Example 2

To prove that the formula

$$F : (P \rightarrow Q) \land (Q \rightarrow R) \rightarrow (P \rightarrow R)$$

is valid, assume that it is invalid and derive a contradiction:

1.	$I \not\models F$	assumption
2.	$I \models (P \rightarrow Q) \land (Q \rightarrow R)$	by 1 and semantics of $\rightarrow$
3.	$I \not\models P \rightarrow R$	by 1 and semantics of $\rightarrow$
4.	$I \models P$	by 3 and semantics of $\rightarrow$
5.	$I \not\models R$	by 3 and semantics of $\rightarrow$
6.	$I \models P \rightarrow Q$	2 and semantics of $\land$
7.	$I \models Q \rightarrow R$	2 and semantics of $\land$

Two cases consider from 6:

1. $I \not\models P$: contradiction with 4.
2. $I \models Q$: two cases to consider from 7:
   1. $I \not\models Q$: contradiction
   2. $I \models R$: contradiction with 5.

Equivalence and Implication

- Two formulas F_1 and F_2 are equivalent

$$F_1 \iff F_2$$

iff $F_1 \leftrightarrow F_2$ is valid, i.e., for all interpretations I, $I \models F_1 \leftrightarrow F_2$.

- Formula F_1 implies formula F_2

$$F_1 \Rightarrow F_2$$

iff $F_1 \rightarrow F_2$ is valid, i.e., for all interpretations I, $I \models F_1 \rightarrow F_2$.

Normal Forms

A normal form of formulae is a syntactic restriction such that for every formula of the logic, there is an equivalent formula in the normal form.

- **Negation Normal Form (NNF)** requires that $\neg, \wedge$, and $\vee$ be the only connectives and that negations appear only in literals: e.g.,

$$\neg(F_1 \wedge F_2) \iff \neg F_1 \vee \neg F_2$$

- **Disjunctive Normal Form (DNF)** requires that formulae be a disjunction of conjunctions of literals:

$$\bigvee_i \bigwedge_j l_{i,j}$$

- **Conjunctive Normal Form (CNF)** requires that formulae be a conjunction of clauses (disjunctions of literals):

$$\bigwedge_i \bigvee_j l_{i,j}$$

Equisatisfiability

- F and F' are equisatisfiable when F is satisfiable iff F' is satisfiable.
  - Equisatisfiability is a weaker notion of equivalence, which is still useful when deciding satisfiability.
- SAT solvers convert a given formula to an equisatisfiable formula in CNF.
  - A formula can be converted to an equisatisfiable formula in CNF with only a linear increase in size (Tseitin's transformation).
  - Conversion to an equivalent CNF incurs exponential blow-up in worst-case.

Decision Procedures

Two approaches for deciding satisfiability:

- **Search**: exhaustively search through all possible assignments:

let rec **SAT** F =
  if $F = \top$ then true
  else if $F = \bot$ then false
  else
    let P = **Choose**(**vars**(F)) in
    (**SAT** $F\{P \mapsto \top\}$) $\vee$ (**SAT** $F\{P \mapsto \bot\}$)

- **Deduction**: iteratively apply proof rules (resolution):

$$\frac{C_1[P] \quad C_2[\neg P]}{C_1[\bot] \vee C_2[\bot]}$$

The Resolution Procedure

$$\frac{C_1[P] \quad C_2[\neg P]}{C_1[\bot] \vee C_2[\bot]}$$

- To satisfy clauses $C_1[P]$ and $C_2[\neg P]$, either the rest of C_1 or the rest of C_2 must be satisfied. If P is true, then a literal other than $\neg P$ in C_2 must be satisfied; while if P is false, then a literal other than P in C_1 must be satisfied.
- If ever $\bot$ is deduced via resolution, F is unsatisfiable. Otherwise, if no further resolutions are possible, F is satisfiable.

Examples

- $(\neg P \vee Q) \wedge P \wedge \neg Q$

  From resolution

  $$\frac{(\neg P \vee Q)}{Q},$$

  construct

  $$(\neg P \vee Q) \wedge P \wedge \neg Q \wedge Q$$

  which derives $\bot$.

- $(\neg P \vee Q) \wedge \neg Q)$

  The resolution procedure yields

  $$(\neg P \vee Q) \wedge \neg Q \wedge \neg P$$

  No further resolutions are possible.

DPLL

The Davis-Putnam-Logemann-Loveland algorithm (DPLL) combines the enumerative search and a restricted form of resolution, called unit resolution:

$$\frac{l \qquad C[\neg l]}{C[\bot]}$$

The process of applying this resolution as much as possible is called Boolean constraint propagation (BCP).

let rec **DPLL** F =
  let F' = **BCP**(F) in
  if $F' = \top$ then true
  else if $F' = \bot$ then false
  else
    let P = **Choose**(**vars**(F')) in
    (**DPLL** $F'\{P \mapsto \top\}$) $\vee$ (**DPLL** $F'\{P \mapsto \bot\}$)

MaxSAT Example: Software Upgradeability Problem[1]

Package	Dependencies	Conflicts
p_1	$\{p_2 \vee p_3\}$	$\{p_4\}$
p_2	$\{p_3\}$	$\emptyset$
p_3	$\{p_2\}$	$\{p_4\}$
p_4	$\{p_2 \wedge p_3\}$	$\emptyset$

- Encoding dependencies:
  - $p_1 \rightarrow (p_2 \vee p_3) \equiv (\neg p_1 \vee p_2 \vee p_3)$
  - $p_2 \rightarrow p_3 \equiv (\neg p_2 \vee p_3)$
  - $p_3 \rightarrow p_2 \equiv (\neg p_3 \vee p_2)$
  - $p_4 \rightarrow (p_2 \wedge p_3) \equiv (\neg p_4 \vee p_2) \wedge (\neg p_4 \vee p_3)$
- Encoding conflicts:
  - $p_1 \rightarrow \neg p_4 \equiv (\neg p_1 \vee \neg p_4)$
  - $p_3 \rightarrow \neg p_4 \equiv (\neg p_3 \vee \neg p_4)$
- Encoding installing all packages:
  - $p_1 \wedge p_2 \wedge p_3 \wedge p_4$

[1] Slides from `http://www.cs.utexas.edu/~isil/cs389L/ut-maxsat.pdf`

Example

The formula in CNF:

$$\neg p_1 \lor p_2 \lor p_3, \quad \neg p_2 \lor p_3, \quad \neg p_3 \lor p_2, \quad \neg p_4 \lor p_2,$$
$$\neg p_4 \lor p_3, \quad \neg p_1 \lor \neg p_4, \quad \neg p_3 \lor \neg p_4$$
$$p_1, \quad p_2, \quad p_3, \quad p_4$$

- The formula is unsatisfiable.
- How many clauses can we satisfy?

Maximum Satisfiability (MaxSAT)

- MaxSat:
  - An optimization extension of SAT.
  - All clauses are soft.
  - Maximize number of satisfied soft clauses.
- Partial MaxSAT:
  - Clauses in the formula are soft or hard.
  - Hard clauses must be satisfied.
  - Maximize number of satisfied soft clauses.
- Weighted Partial MaxSAT:
  - Clauses are soft or hard.
  - Soft clauses are associated with weights.
  - Maximize sum of weights of satisfied clauses.
- MaxSAT has a variety of applications. Any optimization problem is likely to be solved by MaxSAT.

Example: Partial MaxSAT

- Dependencies and conflicts are hard constraints:

$$\neg p_1 \vee p_2 \vee p_3, \quad \neg p_2 \vee p_3, \quad \neg p_3 \vee p_2, \quad \neg p_4 \vee p_2,$$
$$\neg p_4 \vee p_3, \quad \neg p_1 \vee \neg p_4, \quad \neg p_3 \vee \neg p_4$$

- Installation of packages are soft constraints:

$$p_1, \quad p_2, \quad p_3, \quad p_4$$

- Goal: maximize the number of installed packages.
- Optimal solution:

$$p_1 = \top, p_2 = \top, p_3 = \top, p_4 = \bot$$

First-Order Logic

- In FOL, terms evaluate to values other than truth values.
- Terms include variables $x, y, z, \ldots$, constants $a, b, c, \ldots$, and functions $f, g, h, \ldots$.
  - An n-ary function f takes n terms as arguments. E.g., $f(a), g(x, b), f(g(x, f(b)))$.
  - A constant can be viewed as a 0-ary function.
- Propositional variables are generalized to predicates $p, q, r, \ldots$.
  - An n-ary predicate takes n terms as arguments.
  - A propositional variable is a 0-ary predicate: $P, Q, R, \ldots$.
- An atom is $\top, \bot$, or an n-ary predicate applied to n terms.
- A literal is an atom or its negation: e.g., $P, p(f(x), g(x, f(x)))$.

Syntax of First-Order Logic

$$\begin{array}{rcl} F & \rightarrow & \bot \\ & | & \top \\ & | & p(t_1, \ldots, t_n) \\ & | & \neg F \\ & | & F_1 \wedge F_2 \\ & | & F_1 \vee F_2 \\ & | & F_1 \rightarrow F_2 \\ & | & F_1 \leftrightarrow F_2 \\ & | & \forall x. F[x] \\ & | & \exists x. F[x] \end{array}$$

Interpretation

The notion of interpretation is more complicated than PL:

- The domain D_I of an interpretation is a nonempty set of values or objects, such as integers, real numbers, people, etc.
- The assignment α_I of interpretation I maps constant, function, and predicate symbols to elements, functions, and predicates over D_I. It also maps variables to elements of D_I.
  - Each variable symbol x is assigned a value x_I from D_I.
  - Each n-ary function symbol f is assigned an n-ary function

$$f_I : D_I^n \to D_I$$

  - Each n-ary predicate symbol p is assigned an n-ary predicate

$$p_I : D_I^n \to \{\textbf{true}, \textbf{false}\}$$

- An interpretation $I : (D_I, \alpha_I)$ is a pair of a domain and an assignment.

Example

$$F : x + y > z \rightarrow y > z - x$$

- Note $+, -, >$ are just symbols: $p(f(x, y), z) \rightarrow p(y, g(z, x))$.
- Domain:

$$D_I = \mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$$

- Assignment:

$$\alpha_I = \{+ \mapsto +_{\mathbb{Z}}, - \mapsto -_{\mathbb{Z}}, > \mapsto >_{\mathbb{Z}}, x \mapsto 13, y \mapsto 42, z \mapsto 1, \ldots\}$$

Semantics of First-Order Logic

Given an interpretation $I : (D_I, \alpha_I)$, $I \models F$ or $I \not\models F$.

$$
\begin{array}{ll}
I \models \top, \quad I \not\models \bot, & \\
I \models p(t_1, \ldots, t_n) & \text{iff} \;\; \alpha_I[p(t_1, \ldots, t_n)] = \textbf{true} \\
I \models \neg F & \text{iff} \;\; I \not\models F \\
I \models F_1 \wedge F_2 & \text{iff} \;\; I \models F_1 \text{ and } I \models F_2 \\
I \models F_1 \vee F_2 & \text{iff} \;\; I \models F_1 \text{ or } I \models F_2 \\
I \models F_1 \rightarrow F_2 & \text{iff} \;\; I \not\models F_1 \text{ or } I \models F_2 \\
I \models F_1 \leftrightarrow F_2 & \text{iff} \;\; (I \models F_1 \text{ and } I \models F_2) \text{ or } (I \not\models F_1 \text{ and } I \not\models F_2) \\
I \models \forall x.F & \text{iff for all } v \in D_I, I \lhd \{x \mapsto v\} \models F \\
I \models \exists x.F & \text{iff there exists } v \in D_I, I \lhd \{x \mapsto v\} \models F
\end{array}
$$

where $I \lhd \{x \mapsto v\}$ denotes an x-variant of I.

Example

$$F : \exists x. f(x) = g(x)$$

Consider the interpretation $I : (D : \{v_1, v_2\}, \alpha_I)$:

$$\alpha_I : \{f(v_1) \mapsto v_1, f(v_2) \mapsto v_2, g(v_1) \mapsto v_2, g(v_2) \mapsto v_1\}$$

Compute the truth value of F under I as follows:

$$\begin{array}{llcll} 1. & I \lhd \{x \mapsto v\} & \not\models & f(x) = g(x) & \text{for } v \in D \\ 2. & I & \not\models & \exists x. f(x) = g(x) & \text{since } v \in D \text{ is arbitrary} \end{array}$$

Satisfiability and Validity

- A formula F is *satisfiable* iff there exists an interpretation I such that $I \models F$.
- A formula F is *valid* iff for all interpretations I, $I \models F$.
- Satisfiability and validity only apply to closed FOL formulas.
  - If we say that a formula F such that $\mathbf{free}(F) \neq$ is valid, we mean that its universal closure $\forall * .F$ is valid.
  - If we say that F is satisfiable, we mean that its existential closure $\exists * .F$ is satisfiable.
- Duality still holds:

$$\forall * .F \text{ is valid} \iff \exists * .\neg F \text{ is unsatisfiable.}$$

First-Order Theories

- While validity in FOL is undecidable, validity in particular theories or fragments of theories is sometimes decidable.
- A first-order theory T is defined by signatures and axioms:
  - Its signature Σ is a set of constant, function, and predicate symbols.
  - Its set of axioms $\mathcal{A}$ is a set of closed FOL formulas in which only constant, function, and predicate symbols of Σ appear.
- A Σ-formula F is valid in the theory T, or T-valid, if every interpretation I that satisfies the axioms of T,

  $$I \models A \text{ for every } A \in \mathcal{A} \text{ (}I\text{ is a }T\text{-interpretation)}$$

  also satisfies $F : I \models F$. We write $T \models F$ for T-validity of F.
- The theory T consists of all (closed) formulas that are T-valid.
- A Σ-formula F is satisfiable in T, or T-satisfiable, if there is a T-interpretation I that satisfies F.
- The quantifier-free fragment of a theory T is the set of formulas without quantifiers that are valid in T.

The Theory of Equality

- $\Sigma_E : \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r, \ldots\}$
- Axioms $\mathcal{A}$:
  1. $\forall x. x = x$
  2. $\forall x, y. x = y \to y = x$
  3. $\forall x, y, z. x = y \land y = z \to x = z$
  4. for each positive integer n and n-ary function symbol f,

     $$\forall \bar{x}, \bar{y}. (\bigwedge_{i=1}^{n} x_i = y_i) \to f(\bar{x}) = f(\bar{y})$$

  5. for each positive integer n and n-ary predicate symbol p,

     $$\forall \bar{x}, \bar{y}. (\bigwedge_{i=1}^{n} x_i = y_i) \to (p(\bar{x}) \leftrightarrow p(\bar{y}))$$

T_E is undecidable, but the quantifier-free fragment of T_E is decidable.

Example

$$F : a = b \land b = c \rightarrow g(f(a), b) = g(f(c), a)$$

Is F T_E-valid?

Useful First-Order Theories

Theory	**Description**	**Full**	**QFF**
T_{E}	equality	no	yes
T_{PA}	Peano arithmetic	no	no
$T_{\mathbb{N}}$	Presburger arithmetic	yes	yes
$T_{\mathbb{Z}}$	linear integers	yes	yes
$T_{\mathbb{R}}$	reals (with $\cdot$)	yes	yes
$T_{\mathbb{Q}}$	rationals (without $\cdot$)	yes	yes
T_{RDS}	recursive data structures	no	yes
T^{+}_{RDS}	acyclic recursive data structures	yes	yes
T_{A}	arrays	no	yes
$T^{=}_{\mathsf{A}}$	arrays with extensionality	no	yes

- In practice, we want to check for satisfiability span multiple theories, e.g., verifying programs that manipulate integers and a list of reals.
- Nelson-Oppen combination of decision procedures.

Program Verification

Three foundational methods underlying all verification and program analysis techniques:

- **Specification** (program annotation) is the precise statement of properties that a program should exhibit.
- **Inductive assertion method** is for proving partial correctness properties.
- **Ranking function method** is for proving total correctness properties.

Example: Linear Search

```
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while (i ≤ u) {
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Specification (Program Annotations)

- An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs.
- An annotation F at location L asserts that F is true whenever program control reaches L.
- Types of annotations:
  - **Function specification**: precondition + postcondition.
  - **Loop invariant**
  - **Assertion**

Function Specifications

Formulas whose free variables include only the formal parameters and return variables.

- Precondition: Specification about what should be true upon entering the function.
- Postcondition: Specification about the expected output of the function.

Function Specifications

The behavior of LinearSearch:

- It returns true iff the array a contains the value e in the range $[l, u]$.
- It behaves correctly only when $l \geq 0$ and $u < |a|$.

Function specification formalizes these statements:

```
@pre : 0 ≤ l ∧ u < |a|
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while (i ≤ u) {
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Our goal is to prove the *partial correctness* property: if the function precondition holds and the function halts, then the function postcondition holds upon return.

Loop Invariants

For proving partial correctness, each loop must be annotated with a loop invariant $\boldsymbol{F}$:

$$
\begin{array}{l}
\text{while} \\
\quad @\boldsymbol{F} \\
\quad (\langle \boldsymbol{condition} \rangle)\ \{ \\
\quad \langle \boldsymbol{body} \rangle \\
\}
\end{array}
$$

- $\boldsymbol{F}$ holds at the beginning of every iteration.
- $\boldsymbol{F} \wedge \langle \boldsymbol{condition} \rangle$ holds in the body.
- $\boldsymbol{F} \wedge \neg \langle \boldsymbol{condition} \rangle$ holds when exiting the loop.

Loop Invariants

In LinearSearch, whenever control reaches the loop entry (L), the loop index is at least l and that $a[j] \neq e$ for previously examined indices j:

```
@pre : 0 ≤ l ∧ u < |a|
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

cf) Inference of Preconditions and Loop Invariants

Automatic inference of preconditions and loop invariants is an active research area: e.g.,

- Data-driven precondition inference with learned features. PLDI 2016.
- Learning invariants using decision trees and implication counterexamples. POPL 2016.
- A data-driven approach for algebraic loop invariants. ESOP 2013.
- Inductive invariant generation via abductive inference. OOPSLA 2013.
- ...

Abstract interpretation can be viewed as a method for automatically inferring loop invariants.

Assertions

Programmers's formal comments on the program behavior:

$$
\begin{array}{l}
@\text{pre} : 0 \leq l \wedge u < |a| \\
@\text{post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e \\
\text{bool LinearSearch (int } a[], \text{ int } l, \text{ int } u, \text{ int } e) \ \{ \\
\quad \text{int } i := l; \\
\quad \text{while} \\
\quad @L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e) \\
\quad (i \leq u) \ \{ \\
\quad @0 \leq i < |a| \\
\qquad \text{if } (a[i] = e) \text{ return true} \\
\qquad i := i + 1; \\
\quad \} \\
\quad \text{return false} \\
\}
\end{array}
$$

Partial Correctness

- A function is partially correct if when the function's precondition is satisfied on entry, its postcondition is satisfied when the function returns (if it ever does).
- Inductive assertion method:
  - Derive verification conditions (VCs) from a function.
  - Check the validity of VCs by an SMT solver.
  - If all of VCs are valid, the function obeys its specification.

Deriving VCs

Done in two steps:

- The function is broken down into a finite set of *basic paths*.
- Each basic path generates a verification condition.
- Loops complicate proofs as they create unbounded number of paths. For loops, loop invariants cut the paths into a finite set of basic paths.

Basic Paths

- A basic path is a sequence of atomic statements that begins at the function precondition or a loop invariant and ends at a loop invariant or the function postcondition.
- Moreover, a loop invariant can only occur at the beginning or the ending of a basic path (Basic paths do not cross loops).

Program	Basic Paths

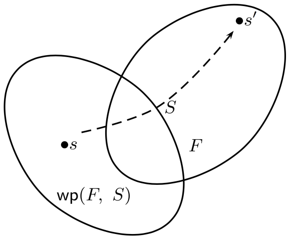
$@\text{pre}: 0 \leq l \land u < \lvert a \rvert$
$@\text{post}: rv \leftrightarrow \exists i.l \leq i \leq u \land a[i] = e$
bool LinearSearch (int a[], int l, int u, int e) {
  int $i := l$;
  while
  $@L: l \leq i \land (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
  $(i \leq u)$ {
    if $(a[i] = e)$ return true
    $i := i + 1$;
  }
  return false
}
	(1)
	$@\text{pre}: 0 \leq l \land u < \lvert a \rvert$
	$i := l;$
	$@L: l \leq i \land (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
	
	(2)
	$@L: l \leq i \land (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
	assume $i \leq u$;
	assume $a[i] = e$;
	$rv :=$ true
	$@\text{post}: rv \leftrightarrow \exists i.l \leq i \leq u \land a[i] = e$
	
	(3)
	$@L: l \leq i \land (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
	assume $i \leq u$;
	assume $a[i] \neq e$
	$i := i + 1;$
	$@L: l \leq i \land (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
	
	(4)
	$@L: l \leq i \land (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
	assume $i > u$;
	$rv :=$ false
	$@\text{post}: rv \leftrightarrow \exists i.l \leq i \leq u \land a[i] = e$

Weakest Precondition Transformer

The reduction from basic paths to verification conditions requires the weakest precondition transformer:

$$\textbf{wp} : \textbf{FOL} \times \textbf{stmts} \rightarrow \textbf{FOL}$$

The weakest precondition $\textbf{wp}(F, S)$ has the defining characteristic that every state s on which executing statement S leads to a state s' in the F region must be in the $\textbf{wp}(F, S)$ region:



For F to hold after executing S, $\textbf{wp}(F, S)$ must hold before executing S.

Weakest Precondition Transformer

Weakest precondition $\textbf{wp}(F, S)$ for statements S of basic paths:

- Assumption: What must hold before statement **assume** c is executed to ensure that F holds afterwards? If $c \rightarrow F$ holds before, then satisfying c guarantees that F holds afterwards:

$$\textbf{wp}(F, \textbf{assume}\ c) \Leftrightarrow c \rightarrow F.$$

- Assignment: What must hold before statement $v := e$ is executed to ensure that $F[v]$ holds afterward? If $F[e]$ holds before, then assigning e to v makes $F[v]$ holds afterward:

$$\textbf{wp}(F, v := e) \Leftrightarrow F[e]$$

For a sequence of statements $S_1; \ldots; S_n$, define

$$\textbf{wp}(F, S_1; \ldots, S_n) \Leftrightarrow \textbf{wp}(\textbf{wp}(F, S_n), S_1; \ldots, S_{n-1}).$$

Verification Conditions

The verification condition of basic path

$$\begin{array}{l} @F \\ S_1; \\ \vdots \\ S_n; \\ @G \end{array}$$

is

$$F \rightarrow \mathbf{wp}(G, S_1; \ldots; S_n)$$

The verification condition is sometimes denoted by the Hoare triple

$$\{F\}S_1; \ldots; S_n\{G\}.$$

Example

$$@L : F : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$$
$$S_1 : \text{assume } i \leq u;$$
$$S_2 : \text{assume } a[i] = e;$$
$$S_3 : rv := \text{true}$$
$$@\text{post}G : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$$

The VC is

$$l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$$
$$\rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists j. l \leq j \leq u \wedge a[j] = e))$$

$$\begin{array}{l}
\mathsf{wp}(G,\ S_1; S_2; S_3) \\
\Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(rv \ \leftrightarrow\ \exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e,\ rv := \mathtt{true}),\ S_1; S_2) \\
\Leftrightarrow\ \mathsf{wp}(\mathtt{true} \ \leftrightarrow\ \exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e,\ S_1; S_2) \\
\Leftrightarrow\ \mathsf{wp}(\exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e,\ S_1; S_2) \\
\Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(\exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e,\ \mathtt{assume}\ a[i] = e),\ S_1) \\
\Leftrightarrow\ \mathsf{wp}(a[i] = e \ \rightarrow\ \exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e,\ S_1) \\
\Leftrightarrow\ \mathsf{wp}(a[i] = e \ \rightarrow\ \exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e,\ \mathtt{assume}\ i \leq u) \\
\Leftrightarrow\ i \leq u \ \rightarrow\ (a[i] = e \ \rightarrow\ \exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e)
\end{array}$$

Total Correctness

Total correctness of a function asserts that if the precondition holds on entry, then the function eventually halts and the postcondition holds.

Well-Founded Relation

A binary relation $\prec$ over a set S is well-founded iff there does not exist an infinite sequence $s_1, s_2, \ldots$ of elements of S such that

$$s_1 \prec s_2 \prec \cdots .$$

For example, the relation $<$ is well-founded over the natural numbers, because any sequence of natural numbers decreasing according to $<$ is finite: e.g.,

$$1023 > 39 > 30 > 29 > 8 > 3 > 0.$$

However, the relation $<$ is not well-founded over the rationals or reals.

Proving Termination

- Define a set S with a well-founded relation $\prec$.
  - We usually choose as S the set of n-tuples of natural numbers and as $\prec_n$ the lexicographic extension[2] of $\prec$, where n varies according to the application.
- Find a *ranking function* δ mapping program states to S such that δ decreases according to $\prec$ along every basic path.
- Then, since $\prec$ is well-founded, there cannot exist an infinite sequence of program states.

[2] When $n = 2$, $(a, b) \prec_2 (a', b') \iff a \prec a' \lor (a = a' \land b \prec b')$

Example

```
@pre ⊤
@post ⊤
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for
    @L_1 : i + 1 ≥ 0
    ↓ (i + 1, i + 1)
    (int i := |a| − 1; i > 0; i := i − 1) {
    for
      @L_2 : i + 1 ≥ 0 ∧ i − j ≥ 0
      ↓ (i + 1, i − j)
      (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```

Verification Conditions

The verification condition of basic path

$$\begin{array}{l} @F \\ \downarrow \delta[\bar{x}] \\ S_1; \\ \vdots \\ S_n; \\ \downarrow \kappa[\bar{x}] \end{array}$$

is

$$F \rightarrow \mathsf{wp}(\kappa \prec \delta[\bar{x}_0], S_1; \ldots; S_n)\{\bar{x}_0 \mapsto \bar{x}\}$$

Example

The verification condition for the basic path

$$\begin{array}{l} @L_1 : i + 1 \geq 0 \\ \downarrow L_1 : (i+1, i+1) \\ \textbf{assume } i > 0; \\ j := 0; \\ \downarrow L_2 : (i+1, i-j) \end{array}$$

is

$$i + 1 \geq 0 \wedge i > 0 \rightarrow (i+1, i-0) <_2 (i+1, i+1).$$