Short Notes on Semantics, Symbolic Evaluation, Bounded Model-checking, VC-generation based Verification, and Hoare Logic

Madhusudan Parthasarathy

September 10, 2026

1 While programs

Let's define while programs over the domain of integers as follows.

Fix a set of variables X of type integer

$$AExp :: -x \mid c \mid AExp + AExp \mid AExp - AExp \mid AExp * AExp$$

$$BExp :: -T \mid F \mid AExp \leq AExp \mid AExp < AExp \mid AExp = AExp \mid BExp \vee BExp \mid BExp \wedge BExp \mid \neg BExp$$

$$S :: -skip \mid x := AExp \mid S; S \mid \texttt{ assume } BExp \mid \texttt{if } (BExp) \texttt{ then } \{S\} \texttt{ else } \{S\} \mid \texttt{while } (BExp) \ \{S\}$$

We also assume in this section that sequential composition of programs is always bracketed right-associatively, so that in any program $S_1; S_2$, S_1 is not of the form $S; S'$.

So for example, $x := 1; x := 2; x := 3$ is parsed only as $x := 1; (x := 2; x := 3)$.

Consequently, whenever we see an expression of the form $S_1; S_2$, we assume S_1 is an assignment statement, skip statement, assume statement, conditional statement, or a while statement. Furthermore, we will *construct* new programs from old below. In these definitions, even if the new program doesn't seem to parse with the above convention, we assume it's automatically rewritten to parse in the above way.

For technical convenience, let us assume in this section that all full programs end with the statement `skip`.

2 Operational Semantics

2.1 Small-step Operational Semantics using Transition Systems

Given a program S over variables X, we define its small-step operational semantics using a transition system. The states of the transition system are tuples of the form $(Store, S)$ where $Store : X \to \mathbb{Z}$, and S is either a program or the special symbol ϵ.

Let us first define the evaluation of arithmetic and Boolean expressions with respect to a store:

$$\begin{aligned} eval(Store, y) &= Store(y), \ \ where\ y \in X \\ eval(Store, c) &= c, \ \ where\ c \in \mathbb{Z} \\ eval(Store, e_1 + e_2) &= eval(Store, e_1) + eval(Store, e_2) \\ eval(Store, e_1 - e_2) &= eval(Store, e_1) - eval(Store, e_2) \\ eval(Store, e_1 * e2) &= eval(Store, e_1) * eval(Store, e_2) \end{aligned}$$

And for Boolean expressions:

$$\begin{aligned}
eval(Store, T) &= T \\
eval(Store, F) &= F \\
eval(Store, e_1 \leq e_2) &= T \ \textit{if } eval(Store, e_1) \leq eval(Store, e_2) \\
&= F \ \textit{otherwise} \\
eval(Store, e_1 < e_2) &= T \ \textit{if } eval(Store, e_1) < eval(Store, e_2) \\
&= F \ \textit{otherwise} \\
eval(Store, e_1 = e_2) &= T \ \textit{if } eval(Store, e_1) = eval(Store, e_2) \\
&= F \ \textit{otherwise} \\
eval(Store, b_1 \vee b_2) &= eval(Store, b_1) \vee eval(Store, b_2) \\
eval(Store, b_1 \wedge b_2) &= eval(Store, b_1) \wedge eval(Store, b_2) \\
eval(Store, \neg b) &= \neg eval(Store, b)
\end{aligned}$$

Transitions are deterministic, as expected, and defined by the following *partial* function δ:

$$\begin{aligned}
\delta(Store,\ \texttt{skip}\ ) &= (Store, \epsilon) \\
\delta(Store,\ \texttt{skip ; S}) &= (Store, S) \\
\delta(Store, x := e;\ S) &= (Store[x \mapsto eval(Store, e)], S) \\
\delta(Store,\ \texttt{assume}\ b; S) &= (Store, S) \quad \textit{if } eval(Store, b) = T \\
\delta(Store,\ \texttt{if b then \{S}_1\texttt{\} else \{S}_2\texttt{\}; S}) &= (Store, S_1; S) \quad \textit{if } eval(Store, b) = T \\
&= (Store, S_2; S) \quad \textit{otherwise} \\
\delta(Store,\ \texttt{while (b)\{S\}; S}') &= (Store, S;\ \texttt{while (b)\{S\}; S}') \quad \textit{if } \texttt{eval(Store, b)} = \texttt{T} \\
&= (Store, S') \quad \textit{otherwise}
\end{aligned}$$

Note that $\delta(Store,\ \texttt{assume}\ b; S)$ is undefined when $eval(Store, b) = F$. Hence δ is a partial function.

We say that for any store $Store$, the store $Store'$ is reachable on a terminating computation of S if the state $(Store', \epsilon)$ is reachable from $(Store, S)$ (using a finite set of transitions as described above). Formally, if there is a sequence of pairs $(Store_0, S_0), (Store_1, S_1), \ldots (Store_n, S_n)$, where $Store_0 = Store$, $S_0 = S$, $Store_n = Store'$, $S_n = \epsilon$, and for each $0 \leq i < n$, $\delta(Store_i, S_i) = (Store_{i+1}, S_{i+1})$.

3 Program Verification expressed as a Hoare triple

Let us fix a logic that can express predicates about the store. These can be quantifier-free logics over store variables or first-order (or higher-order) logics over the store variables as free variables.

A Hoare triple is of the form $\{P\}S\{Q\}$, where P, Q are predicates on stores and S is a program. We say that $\{P\}S\{Q\}$ is a valid Hoare triple if for every store $Store$ such that $Store \models P$, if $Store'$ is reachable on a terminating computation of S from $Store$, then $Store' \models Q$.

4 Basic Paths and Symbolic Evaluation of Bounded Length

4.1 Basic paths of length n of a program

Basic paths are given by the following set of strings of atomic statements:

$$BP :: -\epsilon \mid x := AExp \cdot BP \mid \texttt{ assume } BExp \cdot BP$$

where $AExp$ and $BExp$ are defined as before.

Let $n \in \mathbb{N}$, $n \geq 0$.

Let us define the basic paths of length n of a program S, $BP(n, S)$ that terminates (finishes execution of the program S entirely). Paths that do not terminate executing S will not be in $BP(n, S)$.

$$\begin{aligned}
BP(n, \texttt{ skip }) &= \{\epsilon\} \qquad \textit{for any } n \geq 0 \\
BP(n, \texttt{ skip ;S}) &= BP(n, S) \qquad \textit{for any } n \geq 0 \\
BP(n, x := e; S) &= x := e \ \cdot \ BP(n-1, S) \qquad \textit{for any } n > 0 \\
BP(0, x := e; S) &= \emptyset \\
BP(n, \texttt{ if b then } \{\texttt{S}_1\} \texttt{ else } \{\texttt{S}_2\}; \texttt{S}) &= \texttt{ assume } b \cdot BP(n, S_1; S) \ \cup \ \texttt{ assume } \neg b \cdot BP(n, S_2; S) \\
BP(n, \texttt{ assume } b; S) &= \texttt{ assume } b \cdot BP(n, S) \\
BP(n, \texttt{ while (b)}\{\texttt{S}_1\}; \texttt{S}_2) &= \texttt{ assume } b \cdot BP(n-1, S_1; \texttt{ while (b)}\{\texttt{S}_1\}; \texttt{S}_2) \ \cup \ \texttt{ assume } \neg \texttt{b} \cdot BP(\texttt{n}-1, \texttt{S}_2) \\
BP(0, \texttt{ while (b)}\{\texttt{S}_1\}; \texttt{S}_2) &= \emptyset
\end{aligned}$$

Note that in the above:

- We can think of n as the *fuel*. Fuel gets consumed by each assignment, mainly, but doesn't get consumed on conditionals. However, evaluating conditionals of while loops consumes one unit of fuel (for technical purposes to make recursive definition well-defined).
- The above definition is well defined— recursive calls either reduce the fuel, or reduce the program while keeping the fuel the same. So with lexicographic ordering $(n, |S|)$, the recursion is well-defined. Note that the definition of while loop decreases the fuel but *increases* the size of the program expression.
- When the fuel is 0, and we need fuel for an assignment or a while conditional, the basic paths become empty set. Consequently, recursive calls that concatenate to this set also become empty sets.
- It is easy to show, by induction on n, that every basic path in $BP(n, S)$ has at most n assignments, for any program S.
- For programs that do not have while loops in them, note that there is always some n such that the basic paths of length n capture all paths in the program. We will return to this in a later section.

4.2 Symbolic Evaluation to depth n of a program

Let us fix a program S. Let X be the (finite) set of variables mentioned in S.

We now define the symbolic evaluation of S for depth n, using $BP(n, S)$, defining one disjunct per basic path.

Let X_i, for any $i \in \mathbb{N}$ denote the set of *indexed variables* x_i, for each $x \in X$.

Let's now define a formula, called a *path formula*, $PF(bp)$, for *any basic path*; this is defined as:

$$PF(bp) = PF(0, bp)$$

$$\begin{aligned}
PF(i, \epsilon) &= T \\
PF(i, x := e \cdot bp) &= \ (x_{i+1} = e[\vec{x}_i/\vec{x}]) \ \wedge \bigwedge_{y \in X, y \neq x} (y_{i+1} = y_i) \ \wedge \ PF(i+1, bp) \\
PF(i, assume b \cdot bp) &= b[\vec{x}_i/\vec{x}] \wedge PF(i, bp)
\end{aligned}$$

For any basic path bp with n assignments, it is easy to see that PF(bp) will have variables whose indices range from 0 through n (i.e., $\vec{x}_0, \ldots \vec{x}_n$).

Consequently, for any program S, since paths in $BP(n, S)$ have at most n assignments, for any path bp of S of length n, $PF(bp)$ will have variables whose indices range from 0 through i, where $i \leq n$. Let $len(bp)$ denote the number of assignments in a basic path, which is also the highest index of variables in $PF(bp)$.

We are now ready to define the symbolic formula that captures whether there is an execution of length n that terminates. Let us fix $\vec{x}$ and $\vec{x}'$ as initial and final variables for these paths.

$$SymPF(n, S) = \bigvee_{bp \in BP(n,S), k = len(bp)} (\vec{x} = \vec{x}_0 \quad \wedge \quad PF(bp) \quad \wedge \quad \vec{x'} = \vec{x}_k)$$

The above says that the symbolic path formula of length n of S is the path formulae of each basic path of at most length n that results in a terminating execution of S, and $\vec{x}$ and $\vec{x}'$ are the initial and final valuation of variables along these paths.

Capturing violations of a Hoare triple on paths of length n

Let us fix a program verification task, a Hoare triple $\{P\}S\{Q\}$, where P and Q are some formulas over $\vec{x}$.

We can now write a logical formula that captures that there is a path of execution of length n of S that violates this Hoare triple.

$$Violates(P, S, Q, n) = P(\vec{x}) \wedge SymPF(n, S) \wedge \neg Q[\vec{x}'/\vec{x}]$$

The above formula says that there is a basic path of length n of S that terminates, where the initial variables satisfy P and the final variables do *not* satisfy Q.

If for any n, $Violates(P, S, Q, n)$ is *satisfiable*, then there is clearly a terminating execution of S of length n where the pre-state satisfies P and the terminating state does not satisfy Q, and hence the program does *not* satisfy the Hoare triple specification.

Furthermore, if the Hoare triple is not valid, then there must be some terminating execution that witnesses this violation. If this execution is of length n, then $Violates(P, S, Q, n)$ will be satisfiable.

We hence have the following theorem:

Theorem 4.1 *Let S be a program over variables X, and let $P(\vec{x})$, $Q(\vec{x})$ be state predicates. Then the Hoare triple $\{P\}S\{Q\}$ is not valid iff there exists some $n \in \mathbb{N}$ such that $Violates(P, S, Q, n)$ is satisfiable.*

5 Floyd-style Verification using Inductive Invariants

The verification technique of Section 4 is fundamentally *bounded*: Theorem 4.1 characterizes violations of a Hoare triple, but only by quantifying existentially over the fuel n. It is therefore a method for *finding bugs*, not for *proving correctness*, since a program with a while loop has terminating executions of unboundedly many lengths.

In this section we remove the quantification over n altogether. The idea, due to Floyd, is to ask the programmer to annotate each while loop with an *inductive invariant*: an assertion that holds every time control reaches the header of the loop. Such an annotation lets us *cut* the program at every loop header into finitely many loop-free fragments, each of which can be verified independently, and each of which has only finitely many, boundedly long, executions.

5.1 Annotated while programs

Let us fix an assertion logic in which we write predicates on stores, as in Section 3. We extend the syntax of statements so that every while loop carries an annotation drawn from this logic:

$$S ::= skip \mid x := AExp \mid S; S \mid \texttt{assume}\ BExp \mid \texttt{if}\ (BExp)\ \texttt{then}\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{while}\ (BExp)\ \texttt{Inv}\ I\ \{S\}$$

where I is any formula of the assertion logic over the variables X. We retain all the conventions of Section 1: sequential composition is bracketed right-associatively, and full programs end with the statement `skip`.

Annotations carry *no* semantic content; they are hints supplied by the programmer for the purpose of constructing a proof, and a program means exactly what it meant before the annotation was written. Formally, define the *erasure* $er(S)$ of an annotated program by structural recursion, where the only interesting clause is

$$er(\ \texttt{while}\ (b)\ \texttt{Inv}\ I\ \{S\}) \ = \ \texttt{while}\ (b)\ \{er(S)\}$$

and every other clause simply recurses into the sub-statements. The operational semantics of an annotated program S is by definition that of $er(S)$, and the Hoare triple $\{P\}S\{Q\}$ is valid exactly when $\{P\}er(S)\{Q\}$ is valid in the sense of Section 3. In particular, the annotations may be arbitrary formulas — there is no requirement, built into the syntax, that they be correct or even meaningful. It is the job of the verification conditions below to check that they are.

5.2 Basic blocks with pre/post conditions

A *basic block* is a Hoare triple of the form $\{A\}\,C\,\{B\}$ where A and B are assertions and C is a *basic path*, i.e. $C \in BP$ in the sense of Section 4.1: a string of assignments and assume statements, containing no while loop and no conditional. The point of the construction below is to replace the verification of a single program containing loops by the verification of finitely many basic blocks.

Let $Blk(A, C, S, B)$ denote the set of basic blocks generated by the situation "we started in a store satisfying A, we have so far executed the straight-line code C, the statement S remains to be run, and we are trying to establish B at the end". It is defined by recursion on $|S|$:

$$Blk(A, C,\ \texttt{skip}\ , B) \ = \ \{\ \{A\}\,C\,\{B\}\ \}$$

$$\begin{aligned} Blk(A, C,\ \texttt{skip}\ ; S, B) \ &= \ Blk(A,\ C,\ S,\ B) \\ Blk(A, C, x := e; S, B) \ &= \ Blk(A,\ C \cdot x := e,\ S,\ B) \\ Blk(A, C,\ \texttt{assume}\ b; S, B) \ &= \ Blk(A,\ C \cdot\ \texttt{assume}\ b,\ S,\ B) \end{aligned}$$

$$\begin{aligned} Blk(A, C,\ \texttt{if}\ b\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\}; S, B) \ &= \ Blk(A,\ C \cdot\ \texttt{assume}\ b,\ S_1; S,\ B) \\ &\cup \ \ Blk(A,\ C \cdot\ \texttt{assume}\ \neg b,\ S_2; S,\ B) \end{aligned}$$

$$\begin{aligned} Blk(A, C,\ \texttt{while}\ (b)\ \texttt{Inv}\ I\{S_1\}; S_2, B) \ &= \ \{\ \{A\}\,C\,\{I\}\ \} && \textit{(reaching the header)} \\ &\cup \ \ Blk(I \wedge b,\ \epsilon,\ S_1;\ \texttt{skip}\ ,\ I) && \textit{(the body preserves } I\textit{)} \\ &\cup \ \ Blk(I \wedge \neg b,\ \epsilon,\ S_2,\ B) && \textit{(exiting and continuing)} \end{aligned}$$

The verification condition blocks of a Hoare triple $\{P\}S\{Q\}$ are then simply

$$VC(P, S, Q) \ = \ Blk(P,\ \epsilon,\ S,\ Q).$$

Note the following about this definition:

- Only while loops are *cut*. In the conditional case, the recursion continues on $S_1; S$ and on $S_2; S$: the branch is traversed together with everything that follows the conditional, because there is no assertion available at the join point of an `if-then-else`. At a while loop, on the other hand, the annotation I supplies exactly such an assertion, and so the block being built can be *ended* there (with postcondition I) and new blocks *begun* there (with precondition $I \wedge b$ for the body, and $I \wedge \neg b$ for the continuation), each starting afresh with the empty straight-line code ϵ.

- The invariant I is the only information that crosses a loop header. Everything known about the store before the loop is forgotten on exiting the loop, except what $I \wedge \neg b$ records. In particular, if P says something about a variable that the loop never modifies and that fact is needed after the loop, it must be conjoined into I by hand. This is characteristic of the Floyd discipline and is the usual reason invariants are longer than one first expects.

- The recursion is well-defined: in every clause, the recursive calls are on strictly smaller statements. For the loop, $|S_1;\ \texttt{skip}\ | < |\ \texttt{while}\ (b)\ \texttt{Inv}\ I\{S_1\}; S_2|$ and $|S_2| < |\ \texttt{while}\ (b)\ \texttt{Inv}\ I\{S_1\}; S_2|$. Nested loops need no special treatment; the recursion descends into loop bodies and cuts the loops it finds there in the same way.

- $VC(P, S, Q)$ is a finite set, and every block in it is loop-free and straight-line. If a program has ℓ while loops, the blocks are grouped into $\ell + 1$ families — one per loop header, plus one for the whole program — and within a family there is one block per syntactic path through the loop-free code, so k conditionals in sequence between two cut points yield 2^k blocks.

- If S contains no while loops at all, then $VC(P, S, Q)$ is just the set of blocks $\{P\}\, C\, \{Q\}$ where C ranges over the complete paths of S, recovering the situation of Section 4 with enough fuel.

5.3 Soundness

We now show that verifying all the basic blocks verifies the program. Recall that a store σ' is *reachable on a terminating computation* of a program from σ in the sense of Section 2.1; for a basic path C, which is a program in this sense, this notion applies as well, and since C is straight-line and deterministic, there is at most one such σ'.

Theorem 5.1 (Soundness of the block decomposition) *Let S be an annotated program and let P, Q be assertions. If every basic block in $VC(P, S, Q)$ is a valid Hoare triple, then the Hoare triple $\{P\}S\{Q\}$ is valid.*

Proof. We prove the following more general statement, of which the theorem is the special case $A = P$, $C = \epsilon$, S the whole program, $B = Q$ (taking $\sigma' = \sigma$).

Claim. Let A, B be assertions, C a basic path and S an annotated program, and suppose every block in $Blk(A, C, S, B)$ is valid. Let $\sigma \models A$, let σ' be reachable on a terminating computation of C from σ, and let σ'' be reachable on a terminating computation of $er(S)$ from σ'. Then $\sigma'' \models B$.

The proof is by induction on the length of the terminating computation of $er(S)$ from σ' to σ''. We consider cases on the form of S.

- $S =$ `skip` . Then $\sigma'' = \sigma'$. The block $\{A\}\, C\, \{B\}$ belongs to $Blk(A, C, S, B)$ and is by assumption valid; since $\sigma \models A$ and σ' is reachable on a terminating computation of C from σ, we get $\sigma' \models B$.

- $S = s; S_0$ where s is `skip` , an assignment $x := e$, or `assume` b. The first transition of the computation applies δ to s, producing a store σ_1'; note that in the assume case this transition exists only if $eval(\sigma', b) = T$, so in particular $\sigma_1' = \sigma'$ satisfies b. In each case σ_1' is reachable on a terminating computation of $C \cdot s$ from σ (for $s =$ `skip` , of C itself). The remainder of the computation is a terminating computation of $er(S_0)$ from σ_1' to σ'' and is strictly shorter, and $Blk(A, C \cdot s, S_0, B) = Blk(A, C, S, B)$ consists of valid blocks. The induction hypothesis gives $\sigma'' \models B$.

- $S =$ `if` b `then` $\{S_1\}$ `else` $\{S_2\}; S_0$. Suppose $eval(\sigma', b) = T$; the other case is symmetric. The first transition leaves the store unchanged and continues with $S_1; S_0$, so σ' is reachable on a terminating computation of $C \cdot$ `assume` b from σ, and the remainder is a strictly shorter terminating computation of $er(S_1; S_0)$ from σ' to σ''. Since $Blk(A, C \cdot$ `assume` $b, S_1; S_0, B) \subseteq Blk(A, C, S, B)$, the induction hypothesis applies and gives $\sigma'' \models B$.

- $S =$ `while` (b) `Inv` $I\{S_1\}; S_2$. The block $\{A\}\, C\, \{I\}$ is in $Blk(A, C, S, B)$ and is valid, so $\sigma' \models I$.

  The terminating computation of $er(S)$ from σ' executes the loop body some $k \geq 0$ times: there are stores $\sigma' = \tau_0, \tau_1, \ldots, \tau_k$ such that for each $0 \leq j < k$ we have $eval(\tau_j, b) = T$ and τ_{j+1} is reachable on a terminating computation of $er(S_1)$ from τ_j; further $eval(\tau_k, b) = F$, and σ'' is reachable on a terminating computation of $er(S_2)$ from τ_k. Each of these sub-computations is strictly shorter than the whole.

  We first show $\tau_j \models I$ for every $0 \leq j \leq k$, by induction on j. For $j = 0$ this is $\sigma' \models I$, shown above. Assume $\tau_j \models I$ with $j < k$; since $eval(\tau_j, b) = T$ we have $\tau_j \models I \wedge b$. The blocks $Blk(I \wedge b, \epsilon, S_1;$ `skip` $, I)$ are all valid, being a subset of $Blk(A, C, S, B)$, and τ_{j+1} is reachable on a terminating computation of $er(S_1;$ `skip` $)$ from τ_j, which is a strictly shorter computation. Applying the outer induction hypothesis with $A := I \wedge b$, $C := \epsilon$, $S := S_1;$ `skip` , $B := I$ and $\sigma := \tau_j$ yields $\tau_{j+1} \models I$.

  Hence $\tau_k \models I \wedge \neg b$. The blocks $Blk(I \wedge \neg b, \epsilon, S_2, B)$ are all valid, and σ'' is reachable on a strictly shorter terminating computation of $er(S_2)$ from τ_k; the outer induction hypothesis, with $C := \epsilon$ and $\sigma := \tau_k$, gives $\sigma'' \models B$.

This proves the claim, and with it the theorem. ■

The converse of the theorem does not hold: a programmer may annotate a loop with an assertion that is not invariant, or that is invariant but too weak to establish Q, and then some block will fail even though the program is correct. The verification conditions test the annotations as much as they test the program.

5.4 From basic blocks to logic

It remains to turn each basic block into a logical formula. Since the code C of a block is a basic path, the machinery of Section 4.2 applies to it directly, and — crucially — with no fuel parameter and no disjunction over paths, because a block *is* a single path.

Recall the path formula PF of Section 4.2. It is defined by forward recursion along the path, conjoining at each statement exactly the constraint that statement imposes on the store and advancing the index at each assignment. Let $k = len(C)$, the number of assignments in C, which as noted in Section 4.2 is the highest index occurring in $PF(C)$. Thus $PF(C)$ has free variables among $\vec{x}_0, \ldots, \vec{x}_k$, where $\vec{x}_0$ names the store at the start of the path, $\vec{x}_k$ names the store at its end, and the intermediate copies name the stores in between.

We can now define the *strongest postcondition* of an assertion A along a basic path C: assert A of the initial copy, conjoin the path formula, name the final copy $\vec{x}$ so that the result is again a predicate on a single store, and existentially quantify away every indexed variable.

$$SP(A, C) \;=\; \exists \vec{x}_0 \exists \vec{x}_1 \ldots \exists \vec{x}_k .\; \Big(\; A[\vec{x}_0/\vec{x}] \;\; \wedge \;\; PF(C) \;\; \wedge \;\; \vec{x} = \vec{x}_k \; \Big)$$

where, as in Section 4.2, $\vec{x} = \vec{x}_k$ abbreviates $\bigwedge_{y \in X} (y = y_k)$ and $\exists \vec{x}_i$ abbreviates the block of quantifiers $\exists y_i$ for each $y \in X$. The only free variables of $SP(A, C)$ are $\vec{x}$, so it is an assertion about a single store, as a postcondition should be. Note the similarity to $SymPF$ of Section 4.2: the same device is used to name the initial and final valuations of a path, except that here there is a single path rather than a disjunction over paths, and we keep only the final valuation free.

Observe that this requires the assertion logic to be closed under existential quantification for $SP(A, C)$ to be a formula of the logic; first-order logics are, quantifier-free fragments are not. For many theories of interest, such as linear integer arithmetic, quantifier elimination will return the formula to quantifier-free form. As we will see below, this issue does not in fact arise when we use SP to verify a block.

Lemma 5.2 *Let C be a basic path and $k = len(C)$. For all stores $\sigma, \sigma' : X \to \mathbb{Z}$, the store σ' is reachable on a terminating computation of C from σ if and only if there is a model M of $PF(C)$ with $M(\vec{x}_0) = \sigma$ and $M(\vec{x}_k) = \sigma'$.*

Proof. By induction on C, generalized to $PF(i, C)$ for arbitrary i. For $C = \epsilon$ both sides state that $\sigma' = \sigma$. For $C = x := e \cdot C_0$, the conjuncts added at stage i say exactly that $\vec{x}_{i+1}$ denotes the store $\vec{x}_i[x \mapsto eval(\vec{x}_i, e)]$, which is precisely one application of δ; note that these conjuncts determine $\vec{x}_{i+1}$ uniquely from $\vec{x}_i$. For $C =$ `assume` $b \cdot C_0$, the conjunct $b[\vec{x}_i/\vec{x}]$ holds under M iff $eval(\vec{x}_i, b) = T$, which is exactly the condition under which δ is defined on an assume statement; when it fails, the formula has no model extending $\vec{x}_i$ and equally the computation gets stuck, so the two sides fail together. ■

Since C is deterministic, the model M above is in fact unique given $M(\vec{x}_0)$, so models of $PF(C)$ are in bijection with terminating computations of C.

Lemma 5.3 (*SP* is the strongest postcondition) *Let A be an assertion and C a basic path. Then*

(a) the Hoare triple $\{A\}\, C\, \{SP(A, C)\}$ is valid, and

(b) for every assertion B, if $\{A\}\, C\, \{B\}$ is valid then $SP(A, C) \implies B$ is valid.

Hence $SP(A, C)$ is, up to logical equivalence, the strongest assertion B for which $\{A\}\, C\, \{B\}$ is valid.

Proof. For (a), let $\sigma \models A$ and let σ' be reachable on a terminating computation of C from σ. By Lemma 5.2 there is a model M of $PF(C)$ with $M(\vec{x}_0) = \sigma$ and $M(\vec{x}_k) = \sigma'$. The values $M(\vec{x}_0), \ldots, M(\vec{x}_k)$ witness the existential quantifiers of $SP(A, C)$: the first conjunct holds because $\sigma \models A$, the second because $M \models PF(C)$, and the third because $\sigma' = M(\vec{x}_k)$. Hence $\sigma' \models SP(A, C)$.

For (b), suppose $\sigma' \models SP(A, C)$. Then there are stores $\tau_0, \ldots, \tau_k$ witnessing the quantifiers, and in particular $\tau_0 \models A$, $\tau_k = \sigma'$, and the assignment $\vec{x}_i \mapsto \tau_i$ is a model of $PF(C)$. By Lemma 5.2, σ' is reachable on a terminating computation of C from τ_0. Since $\{A\}\, C\, \{B\}$ is valid and $\tau_0 \models A$, we get $\sigma' \models B$. ■

Theorem 5.4 *Let $\{A\}\, C\, \{B\}$ be a basic block. Then $\{A\}\, C\, \{B\}$ is a valid Hoare triple if and only if*

$$SP(A, C) \implies B$$

is valid.

Proof. The forward direction is Lemma 5.3(b). Conversely, suppose $SP(A, C) \Rightarrow B$ is valid, let $\sigma \models A$ and let σ' be reachable on a terminating computation of C from σ. By Lemma 5.3(a), $\sigma' \models SP(A, C)$, and hence $\sigma' \models B$. ■

Discharging the check without eliminating the quantifiers

Theorem 5.4 says what we are checking, but the formula $SP(A, C) \Rightarrow B$ carries a block of existential quantifiers, which we would rather not hand to a solver. We do not have to. The quantifiers occur *negatively*, in the antecedent of an implication, and so they may simply be dropped and the indexed variables left free.

Indeed, $SP(A, C) \Rightarrow B$ is valid iff its universal closure is, that is, iff

$$\forall \vec{x}\, \forall \vec{x}_0 \ldots \forall \vec{x}_k.\ \Big(A[\vec{x}_0/\vec{x}] \wedge PF(C) \wedge \vec{x} = \vec{x}_k \implies B \Big)$$

holds, using the fact that $(\exists \vec{y}.\varphi) \Rightarrow \psi$ is valid exactly when $\varphi \Rightarrow \psi$ is, provided $\vec{y}$ is not free in ψ. Eliminating $\vec{x}$ using the equality $\vec{x} = \vec{x}_k$, this is in turn equivalent to the validity of the quantifier-free implication over the indexed variables:

$$A[\vec{x}_0/\vec{x}] \ \wedge\ PF(C) \implies B[\vec{x}_k/\vec{x}]$$

or equivalently the *unsatisfiability* of

$$A[\vec{x}_0/\vec{x}] \ \wedge \ PF(C) \ \wedge \ \neg B[\vec{x}_k/\vec{x}].$$

So *SP* tells us *what* a block computes, while the displayed quantifier-free formula is what we actually give the theorem prover. In particular, the assertion logic need not be closed under quantification, and no quantifier elimination is required.

The last formula above is nothing other than $\mathit{Violates}(A, C, B, k)$ of Section 4, specialized to the block: writing $\vec{x}$ for $\vec{x}_0$ and $\vec{x}'$ for $\vec{x}_k$, the disjunction defining $SymPF$ collapses to the single disjunct contributed by the one path C, and the fuel n has been fixed to $k = len(C)$. The same symbolic encoding therefore serves both the bounded model-checking of Section 4 and the Floyd-style proof method here; what the invariant annotations bought us is the elimination of the existential quantification over n in Theorem 4.1.

Collecting the blocks, define

$$\mathit{VCLogic}(P, S, Q) \ = \ \Big\{ \ A[\vec{x}_0/\vec{x}] \wedge PF(C) \implies B[\vec{x}_{len(C)}/\vec{x}] \ \Big| \ \{A\}\, C\, \{B\} \in \mathit{VC}(P, S, Q) \ \Big\}.$$

Corollary 5.4.1 (Floyd-style verification) *Let S be an annotated program and P, Q assertions. If every formula in $\mathit{VCLogic}(P, S, Q)$ is valid, then the Hoare triple $\{P\}S\{Q\}$ is valid.*

Verification has thus been reduced to checking the validity of a *finite* set of formulas, each of which is quantifier-free whenever P, Q and the invariant annotations are, and each of which is discharged by a single call to a theorem prover or SMT solver.

5.5 A remark on the encoding

Frame conjuncts and static single assignment. The conjuncts $\bigwedge_{y \neq x}(y_{i+1} = y_i)$ generated at each assignment are what allow a single global index i to be advanced uniformly for all variables at once: every variable other than the assigned one is explicitly copied forward. This costs $|X| - 1$ equalities per assignment, which is wasteful when X is large. An equivalent encoding gives each variable its own counter, as we worked out in examples in class. This is precisely the *static single assignment* form used by compilers and by real verification-condition generators, and it drops the frame conjuncts entirely. The uniform version above composes more transparently with PF and *Violates* as already defined, which is why we present it this way.