

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

Madhusudan Parthasarathy

September 29, 2026

1 while programs

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

Fix a countably infinite set of variables X of type integer

$$\begin{aligned} AExp &:: -x \mid c \mid AExp + AExp \mid AExp - AExp \mid AExp * AExp \\ BExp &:: -T \mid F \mid AExp \leq AExp \mid AExp < AExp \mid AExp = AExp \mid BExp \vee BExp \mid BExp \wedge BExp \mid \neg BExp \\ S &:: -skip \mid x := AExp \mid S; S \mid \text{assume } BExp \mid \text{if } (BExp) \text{ then } \{S\} \text{ else } \{S\} \mid \text{while } (BExp) \{S\} \end{aligned}$$

In the above, x ranges over X and $c \in \mathbb{Z}$ are arbitrary integers.

Right-associative parsing

We assume in many places in this article 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 also assume in this section that all full programs end with the statement `skip`.

Formally, we can hence think of the grammar of programs as this instead:

$$\begin{aligned} S &:: - S'; skip \\ S' &:: - skip; S' \mid x := AExp; S' \mid \text{assume } BExp; S' \mid \text{if } (BExp) \text{ then } \{S\} \text{ else } \{S\} \mid \text{while } (BExp) \{S\} \end{aligned}$$

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 $Store : X \rightarrow \mathbb{Z}$, 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:

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

And for Boolean expressions:

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

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

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

Note that $\delta(Store, \text{assume } b; S)$ 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 $(Store_0, S_0), (Store_1, S_1), \dots, (Store_n, S_n)$, 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. All we need for these predicates is that for any store $Store$

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:

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

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 $BP(n, S)$.

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

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:

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

$$PF(i, \epsilon) = T$$

$$PF(i, x := e \cdot bp) = (x_{i+1} = e[\vec{x}_i/\vec{x}]) \wedge \bigwedge_{y \in X, y \neq x} (y_{i+1} = y_i) \wedge PF(i+1, bp)$$

$$PF(i, \text{assume } b \cdot bp) = b[\vec{x}_i/\vec{x}] \wedge PF(i, bp)$$

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., $\vec{x}_0, \dots, \vec{x}_n$).

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.

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

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:

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

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

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

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 $|S|$:

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

$$\text{Blk}(A, C, \text{skip}; S, B) = \text{Blk}(A, C, S, B)$$

$$\text{Blk}(A, C, x := e; S, B) = \text{Blk}(A, C \cdot x := e, S, B)$$

$$\text{Blk}(A, C, \text{assume } b; S, B) = \text{Blk}(A, C \cdot \text{assume } b, S, B)$$

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

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

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

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

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, $|S_1; \text{skip}| < |\text{while } (b) \text{ Inv } I \{S_1\}; S_2|$ and $|S_2| < |\text{while } (b) \text{ Inv } I \{S_1\}; S_2|$. 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 .

- $S = \text{skip}$. 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 $s = \text{skip}$, 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 $\sigma'' \models B$.
- $S = \text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\}; S_0$. 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 $C \cdot \text{assume } b$ from σ , and the remainder is a strictly shorter terminating computation of $er(S_1; S_0)$ from σ' to σ'' . Since $Blk(A, C \cdot \text{assume } b, S_1; S_0, B) \subseteq Blk(A, C, S, B)$, the induction hypothesis applies and gives $\sigma'' \models B$.
- $S = \text{while } (b) \text{ Inv } I\{S_1\}; S_2$. 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 $\sigma' = \tau_0, \tau_1, \dots, \tau_k$ 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 $Blk(I \wedge b, \epsilon, S_1; \text{skip}, I)$ are all valid, being a subset of $Blk(A, C, S, B)$, and τ_{j+1} is reachable on a terminating computation of $er(S_1; \text{skip})$ from τ_j , which is a strictly shorter computation. Applying the outer induction hypothesis with $A := I \wedge b$, $C := \epsilon$, $S := S_1; \text{skip}$, $B := I$ 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 $k = \text{len}(C)$, 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 $\vec{x}_0, \dots, \vec{x}_k$, 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.

$$SP(A, C) = \exists \vec{x}_0 \exists \vec{x}_1 \dots \exists \vec{x}_k. \left(A[\vec{x}_0/\vec{x}] \wedge PF(C) \wedge \vec{x} = \vec{x}_k \right)$$

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 $y_i \in X$. 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 $k = \text{len}(C)$. For all stores $\sigma, \sigma' : X \rightarrow \mathbb{Z}$, 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 $\vec{x}_i[x \mapsto \text{eval}(\vec{x}_i, e)]$, which is precisely one application of δ ; note that these conjuncts determine $\vec{x}_{i+1}$ uniquely from $\vec{x}_i$. For $C = \text{assume } b \cdot C_0$, the conjunct $b[\vec{x}_i/\vec{x}]$ holds under M iff $\text{eval}(\vec{x}_i, b) = T$, 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 $M(\vec{x}_0), \dots, M(\vec{x}_k)$ 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 $\tau_0, \dots, \tau_k$ 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 $SP(A, C) \implies B$ 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 $SP(A, C) \implies B$ 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, $SP(A, C) \implies B$ is valid iff its universal closure is, that is, iff

$$\forall \vec{x} \forall \vec{x}_0 \dots \forall \vec{x}_k. \left(A[\vec{x}_0/\vec{x}] \wedge PF(C) \wedge \vec{x} = \vec{x}_k \implies B \right)$$

holds, using the fact that $(\exists \vec{y}. \varphi) \implies \psi$ is valid exactly when $\varphi \implies \psi$ 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 $Violates(A, C, B, k)$ 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

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

Corollary 5.4.1 (Floyd-style verification) *Let S be an annotated program and P, Q assertions. If every formula in $VCLogic(P, S, Q)$ 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.

6 Programs with Methods, Contracts, and Calls

So far a program has been a single block of code manipulating a single store. We now add *methods*: named, separately specified pieces of code that can be called, that take arguments by value, and that — as in Dafny — may return several values at once.

The general inductive invariant method is to argue that there is a single invariant maintained by a system that ensures a set of Unsafe states is never reached. However, this is not precisely the method we follow for proving programs with function calls correct. One reason is that we will not be able to state the global invariant in the code as we did for while programs without function calls, because all variables may not be in scope. Second, the actual runtime configuration of a program with function calls includes the *call stack* (a stack of variables that record the local variables of all methods that are waiting to be resumed), and stating invariants about the call stack is awkward at best and a nightmare at worst. The third reason is that the alternative approach we will use is *modular* in that the methods are proved independent of each other against its *contract*— and this notion of contract goes well beyond just program verification as it serves as a general mode of modular specification for programs that enables local reasoning (testing, verification, monitoring, etc.)

The key idea is *modularity*. We will not verify a call by inlining the body of the callee; inlining after all doesn't even work for when methods call each other recursively. Instead every method carries a *contract*: a precondition the caller must establish, and a postcondition the caller may assume. Verification then becomes a collection of independent tasks, one per method, in which a call is replaced by its contract.

The soundness argument of this verification is a bit more subtle than the usual inductive invariant argument. We present the semantics of programs with method calls in this section as a *proof system* rather than as a transition system, and the height of a derivation in that proof system is exactly the induction measure that can be used to ensure the reasoning is sound.

face

6.1 Syntax

Methods. A method declaration has the form

$$\begin{array}{l} \text{method } m(x_1, \dots, x_k) \text{ returns } (y_1, \dots, y_l) \\ \quad \text{requires } P_m \\ \quad \text{ensures } Q_m \\ \quad \{ S_m \} \end{array}$$

Here $\vec{x}^m = x_1, \dots, x_k$ are the *in-parameters*, $\vec{y}^m = y_1, \dots, y_l$ are the *out-parameters*, P_m is the *precondition*, Q_m is the *postcondition*, and S_m is the *body*. All variables are of type integer, as before. We write X_m for the finite set of variables of m : the in-parameters, the out-parameters, and any other local variables used by S_m . Different methods have disjoint variable sets; there are no global variables.

A *program* Π is a finite set of method declarations with distinct names, together with a distinguished *main* statement S_{main} over its own variable set X_{main} . Recursion and mutual recursion are permitted.

Calls. The statement grammar of Section 5 is extended with one new production:

$$S :: - \dots \mid (z_1, \dots, z_l) := m(e_1, \dots, e_k)$$

where m is a method of Π with k in-parameters and l out-parameters, each e_i is an *AExp* over the variables of the *calling* method, and $z_1, \dots, z_l$ are *distinct* variables of the calling method. As in Section 1, a call occurring in a sequential composition is written $(\vec{z}) := m(\vec{e}); S$.

Immutable in-parameters, and the old construct. Parameters are passed by value, so the body of m receives its own copy of each argument. We adopt Dafny's discipline and forbid the body from assigning

to that copy: the in-parameters are *immutable*. A body that wants to modify an argument copies it into a local first.

The reason is that a postcondition mentioning x_i would otherwise be ambiguous. Does x_i denote the value the caller passed in, or the value m 's local copy happens to hold when the body finishes? Only the first is of any use to a caller — the second is a value the caller cannot observe, since the copy is discarded when the call returns. Immutability removes the ambiguity by making the two coincide.

Specifications are nevertheless written in the usual way, with

old(e) denoting the value of e in the pre-state of the call.

Under our discipline this is no restriction on where *old* may be written and no addition to the logic: since the in-parameters are the only pre-state a postcondition may mention, and they do not change, *old(e)* and e denote the same value, and *old* may simply be erased. We keep the notation because it is standard, and because it stops being vacuous the moment the language acquires state that survives a call — a global variable, a heap, or an in-out parameter. At that point *old* must be given real content: one introduces a *frozen* copy of the mutable pre-state, a postcondition becomes a predicate on a *pair* of stores rather than on one, and the verification conditions must carry the frozen copy along. None of that is needed here, and we say so explicitly rather than leaving the reader to wonder what *old* is doing.

Well-formedness. We require of every method m of Π :

- S_m mentions only variables of X_m , and never assigns to an in-parameter.
- P_m is a formula over $\vec{x}^m$ only.
- Q_m is a formula over $\vec{x}^m \cup \vec{y}^m$ only: the in-parameters, which by immutability still hold the argument values, and the final values of the out-parameters.
- Every call in Π matches the arity of the callee, and its out-targets $z_1, \dots, z_l$ are pairwise distinct.
- *No out-target of a call occurs in any of its arguments:* for a call $(\vec{z}) := m(\vec{e})$, no z_j occurs in any e_i .
- The conventions of Section 1 are retained: sequential composition is right-associated and bodies end with **skip**. While loops carry invariant annotations as in Section 5.

The restriction on Q_m is what makes a contract usable at a call site: a caller can see the arguments it passed and the values it received back, and nothing else of the callee's store.

The last condition — no out-target inside an argument — rules out such calls as $(x) := \text{Incr}(x)$. Section 6.5 explains why: the returned values and the argument values are read at the same point of the generated basic path, so they must not be stored in the same variable. Nothing is lost by it.

Lemma 6.1 (Normal form) *Every call can be brought into the above form. Replacing $(\vec{z}) := m(\vec{e})$ by*

$$a_1 := e_1; \dots; a_k := e_k; (\vec{z}) := m(a_1, \dots, a_k)$$

for fresh local variables $\vec{a}$ of the calling method preserves the semantics of Section 6.2, and the resulting call satisfies the condition.

Proof. The a_i are fresh, so the assignments change nothing that is later read, and $\text{eval}(\sigma, e_i)$ equals the value of a_i when the call is reached; rule (CALL) depends on the arguments only through those values. Freshness also gives $\vec{a} \cap \vec{z} = \emptyset$. ■

So the condition costs no expressiveness: it relocates the temporaries from the verification-condition generator into the program text, where they are assigned by genuine statements and therefore need no justification at all.

6.2 Semantics as a proof system

Since a method is entered with a fresh store, and since the store of a caller and the store of a callee coexist during a call, the single-store transition system of Section 2.1 is no longer the most convenient presentation. We give instead a *big-step* (natural) semantics as a proof system deriving judgements

$$\langle S, \sigma \rangle \Downarrow \sigma'$$

read: “executing S from store σ can terminate in store σ' ”. Here σ and σ' are stores over X_m for the method m in which S occurs. The rules are relative to the fixed program Π . Nothing below mentions contracts or *old*: this is the semantics of the *programming* language, and it is the semantics of Section 2 with one rule added.

$$\begin{array}{c} \frac{}{\langle \text{skip}, \sigma \rangle \Downarrow \sigma} \text{ (SKIP)} \quad \frac{}{\langle x := e, \sigma \rangle \Downarrow \sigma[x \mapsto \text{eval}(\sigma, e)]} \text{ (ASSIGN)} \quad \frac{\text{eval}(\sigma, b) = T}{\langle \text{assume } b, \sigma \rangle \Downarrow \sigma} \text{ (ASSUME)} \\[10pt] \frac{\langle S_1, \sigma \rangle \Downarrow \sigma_1 \quad \langle S_2, \sigma_1 \rangle \Downarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \Downarrow \sigma'} \text{ (SEQ)} \\[10pt] \frac{\text{eval}(\sigma, b) = T \quad \langle S_1, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\}, \sigma \rangle \Downarrow \sigma'} \text{ (IF-T)} \quad \frac{\text{eval}(\sigma, b) = F \quad \langle S_2, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\}, \sigma \rangle \Downarrow \sigma'} \text{ (IF-F)} \\[10pt] \frac{\text{eval}(\sigma, b) = F}{\langle \text{while } (b) \text{ Inv } I\{S\}, \sigma \rangle \Downarrow \sigma} \text{ (WHILE-F)} \\[10pt] \frac{\text{eval}(\sigma, b) = T \quad \langle S, \sigma \rangle \Downarrow \sigma_1 \quad \langle \text{while } (b) \text{ Inv } I\{S\}, \sigma_1 \rangle \Downarrow \sigma'}{\langle \text{while } (b) \text{ Inv } I\{S\}, \sigma \rangle \Downarrow \sigma'} \text{ (WHILE-T)} \\[10pt] \frac{v_i = \text{eval}(\sigma, e_i) \text{ and } \iota(x_i^m) = v_i \text{ for } 1 \leq i \leq k \quad \langle S_m, \iota \rangle \Downarrow \tau}{\langle (\vec{z}) := m(\vec{e}), \sigma \rangle \Downarrow \sigma[z_1 \mapsto \tau(y_1^m), \dots, z_l \mapsto \tau(y_l^m)]} \text{ (CALL)} \end{array}$$

Rule (CALL) says only this: evaluate the arguments in the caller’s store, bind them to the in-parameters of a fresh store for the callee, run the body, and copy the final values of the out-parameters into the caller’s targets.

In the rule, ι ranges over *all* stores on X_m whose in-parameters carry the argument values $\vec{v}$; the values of the other local variables of m are arbitrary. This is the only source of nondeterminism in the language: a local variable that is read before it is written can hold anything, so a verified method may not rely on its initial value. (Dafny instead rejects such programs by a definite-assignment analysis; we choose the semantic route, which costs us nothing below.)

Because $\Downarrow$ is a relation defined by a proof system, every judgement $\langle S, \sigma \rangle \Downarrow \sigma'$ that holds does so by virtue of a finite derivation tree, and we may induct on the *height* of that tree. Note that in (CALL) the premise $\langle S_m, \iota \rangle \Downarrow \tau$ is a strictly shorter derivation than the conclusion, even when m calls itself. This is the observation on which all of Section 6.6 rests.

6.3 Contracts, and what it means for a program to be correct

Because in-parameters are immutable, no generalization of Section 3’s notion of a Hoare triple is required. We record the fact that makes this work.

Write $\text{asgn}(S)$ for the set of variables that S may write: the left-hand side of an assignment, the out-targets $\vec{z}$ of a call, and the union over sub-statements. Well-formedness says $\text{asgn}(S_m) \cap \vec{x}^m = \emptyset$.

Lemma 6.2 (Preservation) *If $\langle S, \sigma \rangle \Downarrow \sigma'$ then $\sigma'(u) = \sigma(u)$ for every variable $u \notin \text{asgn}(S)$. In particular, if $\langle S_m, \iota \rangle \Downarrow \tau$ then $\tau(\vec{x}^m) = \iota(\vec{x}^m)$: a method returns with its in-parameters still holding the argument values.*

Proof. By induction on the height of the derivation. Every rule writes only variables in $\text{asgn}(S)$; in (CALL) the caller’s store is altered only at $\vec{z}$, and the callee runs in a store of its own. ■

Contracts and correctness. A Hoare triple $\{A\}S\{B\}$ over a method m is valid if for every store σ on X_m with $\sigma \models A$ and every σ' with $\langle S, \sigma \rangle \Downarrow \sigma'$ we have $\sigma' \models B$; this is Section 3’s definition verbatim, with $\Downarrow$ in place of reachability. The method m *satisfies its contract* if the triple

$$\{ P_m \} S_m \{ Q_m \}$$

is valid, and the program Π is *correct with respect to* a main specification $\{P\}S_{main}\{Q\}$ if every method of Π satisfies its contract and the triple $\{P\}S_{main}\{Q\}$ is valid.

By Lemma 6.2 this says what it should. The postcondition Q_m is evaluated in the final store τ , but its in-parameters still carry the values the caller supplied, so $\tau \models Q_m$ is a statement relating the arguments to the results — which is exactly what a caller needs and all it can use. Were the in-parameters mutable, this would fail, and a genuinely two-state reading of Q_m would be forced; see the remark on *old* in Section 6.1.

Note that this definition of correctness refers only to the semantics of Section 6.2. No verification machinery — no havoc, no auxiliary variable — occurs in it. Those are devices of the *proof*, introduced next.

6.4 Basic paths with havoc

A call cannot appear on a basic path, since a basic path is by definition a straight-line string of atomic statements. To eliminate calls we need one new atomic statement, which sets a variable to an arbitrary value:

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

Operationally, $\langle \text{havoc } x, \sigma \rangle \Downarrow \sigma[x \mapsto v]$ for every $v \in \mathbb{Z}$. We extend $len(bp)$ to count havocs as well as assignments, and we write $\text{havoc } \vec{z}$ for $\text{havoc } z_1 \dots \text{havoc } z_l$.

Havoc is how a verifier says “the value of x after this point is whatever the callee chose to make it, constrained only by the postcondition”. It is the first genuinely nondeterministic construct in these notes, and it is worth noting where its nondeterminism does and does not matter:

- A basic path containing a havoc no longer determines its final store from its initial store. The remark following Lemma 5.2 — that models of $PF(C)$ are in *bijection* with terminating computations of C — therefore weakens to a correspondence that is onto but not one-to-one. Lemma 5.2 itself, and everything derived from it, is unaffected: nowhere did we use determinism.
- Consequently SP remains the strongest postcondition, Theorem 5.4 still characterizes validity of a block, and the quantifier-free encoding of Section 5.4 remains the formula we hand to the solver.

6.5 Verification condition generation

Fix a program Π . Verification of the method m takes place over its own variable set X_m ; no auxiliary or otherwise extra variable is needed anywhere.

We generate basic blocks exactly as in Section 5.2, with one additional clause of Blk for the call statement. Let $(\vec{z}) := m(\vec{e})$ be a call, and abbreviate by C_{call} the path C extended by the call:

$$C_{call} = C \cdot \text{havoc } \vec{z} \cdot \text{assume } Q_m[\vec{e}/\vec{x}^m, \vec{z}/\vec{y}^m].$$

Then

$$\begin{aligned} Blk(A, C, (\vec{z}) := m(\vec{e}); S, B) &= \{ \{A\} C \{P_m[\vec{e}/\vec{x}^m]\} \} && \text{(the call is legal)} \\ &\cup Blk(A, C_{call}, S, B) && \text{(the call is used)} \end{aligned}$$

Read the clause left to right. One block is *closed* at the call, with postcondition $P_m[\vec{e}/\vec{x}^m]$: this is the obligation that the caller establishes the callee’s precondition, and it is checked in the state just before the call, where the arguments still have their intended values. The path then *continues* as C_{call} — the call is not a cut point — with the out-targets havocked, since the caller knows nothing about them beyond what the

contract says, and with the contract's postcondition assumed, its in-parameters instantiated to the argument expressions and its out-parameters to the actual targets.

The verification conditions of Π against a main specification $\{P\}S_{main}\{Q\}$ are then

$$VC(\Pi, P, Q) = Blk(P, \epsilon, S_{main}, Q) \cup \bigcup_{m \in \Pi} Blk(P_m, \epsilon, S_m, Q_m).$$

Several points deserve emphasis.

- **The construction is modular.** The body S_m of a method contributes blocks that mention no other body — only the *contracts* of the methods it calls. Change a body without changing its contract and only that method's blocks need re-checking.
- **A call is not a cut point.** At a loop header we deliberately forget everything except the invariant, and the block ends there. At a call we must not forget anything: the caller's other variables are untouched by the callee, and facts about them have to survive. So the call contributes a *side* obligation while the path carries on. Havocking exactly $\vec{z}$ is what expresses “the callee changed these and nothing else”; it is the frame condition, and it is free here only because there are no globals and no heap.
- **This is what the restriction on out-targets is for.** In C_{call} the arguments $\vec{e}$ are read *after* `havoc  $\vec{z}$` , and the well-formedness condition of Section 6.1 is exactly what guarantees that this is the same as reading them before it: no e_i mentions a havocked variable, so the frame conjuncts of the havoc carry its value across unchanged.

Drop the condition and the encoding becomes unsound in a way that is easy to miss. Write $x := x + 1$ as $(x) := Incr(x)$, where *Incr* has **ensures** $y = old(x) + 1$, i.e. $y = x + 1$. Substituting the argument x for the in-parameter x of *Incr* gives the path $C \cdot \text{havoc } x \cdot \text{assume } (x = x + 1)$, whose assume is *unsatisfiable*: the path silently disappears, and every block continuing past the call is vacuously valid. A verifier built this way reports success on arbitrary code. By Lemma 6.1 the programmer can always restore the intended meaning by hand, writing $a := x$; $(x) := Incr(a)$.

- **Recursion needs no special treatment.** The clause above uses m 's contract even when the call to m occurs inside S_m itself. So every method is verified assuming its own contract. That this is sound — and not circular — is the content of Theorem 6.3, and the reason it is sound is that the proof inducts on the height of a derivation, which strictly decreases at a call.
- **Only partial correctness is established.** A method that never returns satisfies every postcondition vacuously, and the argument above is quite happy with `method  $m()$  returns () ensures  $F$  {() :=  $m()$ ; skip}`. Proving termination requires ranking functions on calls and on loops, which we do not treat here.

6.6 Soundness

Theorem 6.3 (Soundness of modular verification) *Let Π be a well-formed program and $\{P\}S_{main}\{Q\}$ a specification. If every basic block in $VC(\Pi, P, Q)$ is a valid Hoare triple, then Π is correct with respect to $\{P\}S_{main}\{Q\}$: every method satisfies its contract, and the main triple is valid.*

Proof. Assume every block in $VC(\Pi, P, Q)$ is valid. We prove, by strong induction on $h \in \mathbb{N}$, the following claim.

Claim(h). Let A, B be assertions, C a basic path and S a statement of some method m , all over X_m , and suppose every block in $Blk(A, C, S, B)$ is valid. Let σ be a store on X_m with $\sigma \models A$, let σ' be reachable from σ on a terminating computation of C , and let $\langle S, \sigma' \rangle \Downarrow \sigma''$ have a derivation of height at most h . Then $\sigma'' \models B$.

Assume $\text{Claim}(h')$ for all $h' < h$, and consider a derivation of height at most h . The cases for `skip`, `assignment`, `assume`, `conditional` and `while` are exactly as in the proof of Theorem 5.1, with “strictly shorter computation” replaced throughout by “strictly smaller derivation height”; in each of those cases the premises of the applicable rule are derivations of height less than h , and the blocks needed are, as before, a subset of $\text{Blk}(A, C, S, B)$. We give the one new case.

Case $S = ((\vec{z}) := m'(\vec{e})); S_0$. The derivation ends in (CALL) followed by the rules for S_0 ; more precisely, by (SEQ), there is σ_1 with $\langle (\vec{z}) := m'(\vec{e}), \sigma' \rangle \Downarrow \sigma_1$ and $\langle S_0, \sigma_1 \rangle \Downarrow \sigma''$, both of height less than h . Inverting (CALL) on the first, there are $\vec{v}$ with $v_i = \text{eval}(\sigma', e_i)$, a store ι on $X_{m'}$ with $\iota(\vec{x}^{m'}) = \vec{v}$, and a derivation of $\langle S_{m'}, \iota \rangle \Downarrow \tau$ of height less than h , such that $\sigma_1 = \sigma'[\vec{z} \mapsto \tau(\vec{y}^{m'})]$.

The callee’s precondition holds. The block $\{A\} C \{P_{m'}[\vec{e}/\vec{x}^{m'}]\}$ lies in $\text{Blk}(A, C, S, B)$ and is valid, and σ' is reachable from σ along C , so $\sigma' \models P_{m'}[\vec{e}/\vec{x}^{m'}]$. As $\text{eval}(\sigma', e_i) = v_i = \iota(x_i^{m'})$ and $P_{m'}$ mentions only $\vec{x}^{m'}$, we get $\iota \models P_{m'}$.

The callee’s postcondition holds. The blocks $\text{Blk}(P_{m'}, \epsilon, S_{m'}, Q_{m'})$ are all valid, being part of $\text{VC}(\Pi, P, Q)$, and the derivation of $\langle S_{m'}, \iota \rangle \Downarrow \tau$ has height less than h . So Claim of that smaller height applies with $A := P_{m'}$, $C := \epsilon$, $S := S_{m'}$, $B := Q_{m'}$ and $\sigma := \sigma' := \iota$, giving $\tau \models Q_{m'}$. *This is the step at which a recursive call is discharged, and the only thing that makes it legitimate is that the callee’s derivation is strictly shorter.*

The path extends. We claim σ_1 is itself reachable from σ on a terminating computation of C_{call} . Indeed, σ' is reached along C ; the havoc of the distinct variables $\vec{z}$ may set them to $\tau(\vec{y}^{m'})$, producing exactly $\sigma_1 = \sigma'[\vec{z} \mapsto \tau(\vec{y}^{m'})]$; and the final assume is enabled. For the last point, note that no z_j occurs in any e_i , by the well-formedness condition of Section 6.1, so $\text{eval}(\sigma_1, e_i) = \text{eval}(\sigma', e_i) = v_i$. Since $Q_{m'}$ has free variables among $\vec{x}^{m'} \cup \vec{y}^{m'}$ only, the substituted formula $Q_{m'}[\vec{e}/\vec{x}^{m'}, \vec{z}/\vec{y}^{m'}]$ is evaluated at σ_1 under the valuation $\vec{x}^{m'} \mapsto \vec{v}$ and $\vec{y}^{m'} \mapsto \tau(\vec{y}^{m'})$. By Lemma 6.2, $\tau(\vec{x}^{m'}) = \iota(\vec{x}^{m'}) = \vec{v}$, so that valuation is exactly τ restricted to the free variables of $Q_{m'}$, and the assume holds exactly when $\tau \models Q_{m'}$ — which we have just proved. *This is where immutability of the in-parameters is used, and it is the only place it is needed.*

Conclusion. Every block in $\text{Blk}(A, C_{\text{call}}, S_0, B)$ is valid, being a subset of $\text{Blk}(A, C, S, B)$; $\sigma \models A$; σ_1 is reachable from σ along C_{call} ; and $\langle S_0, \sigma_1 \rangle \Downarrow \sigma''$ has height less than h . The induction hypothesis gives $\sigma'' \models B$.

This completes the induction. Taking $A := P$, $C := \epsilon$, $S := S_{\text{main}}$, $B := Q$ gives validity of the main triple. Taking, for each method m , $A := P_m$, $C := \epsilon$, $S := S_m$, $B := Q_m$ gives validity of $\{P_m\} S_m \{Q_m\}$, that is, m satisfies its contract. ■

As in Section 5, the converse fails, and now for a second reason as well: besides weak or incorrect loop invariants, a contract may be too weak to prove what its callers need, or too strong for its own body to establish. The verification conditions test the contracts exactly as much as they test the code.

6.7 From basic blocks to logic

The blocks produced by $\text{VC}(\Pi, P, Q)$ are ordinary basic blocks whose paths may now contain havocs, so it remains only to extend the path formula PF of Section 4.2 by one clause:

$$PF(i, \text{havoc } x \cdot bp) = \bigwedge_{y \in X, y \neq x} (y_{i+1} = y_i) \wedge PF(i+1, bp)$$

The clause is the assignment clause with the defining equation deleted: the frame is carried forward, and x_{i+1} is left *unconstrained*. That is exactly havoc, and it costs the encoding nothing — indeed a havoc is cheaper than an assignment. In the static-single-assignment presentation mentioned at the end of Section 5 it is cheaper still: a havoc simply increments the counter of x and emits no conjunct at all.

Here X is the variable set over which the block is written, namely X_m for a block generated from the body of m . No variable beyond the locals of the method appears: the frame conjuncts of the havoc carry the in-parameters forward unchanged, which is what makes it legitimate to read the argument expressions after the havoc.

With that single clause, everything in Section 5.4 applies verbatim. For a block $\{A\} C \{B\}$ with $k = \text{len}(C)$,

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

is still the strongest postcondition of A along C (Lemma 5.3 nowhere used determinism), the block is valid iff $SP(A, C) \Rightarrow B$ is valid (Theorem 5.4), and the existentials may again be dropped because they occur in the antecedent, leaving the quantifier-free verification condition

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

Collecting these over all blocks, define

$$VCLogic(\Pi, P, Q) = \left\{ A[\vec{x}_0/\vec{x}] \wedge PF(C) \Longrightarrow B[\vec{x}_{\text{len}(C)}/\vec{x}] \mid \{A\} C \{B\} \in VC(\Pi, P, Q) \right\}.$$

Corollary 6.3.1 (Modular Floyd-style verification) *Let Π be a well-formed program and $\{P\} S_{\text{main}} \{Q\}$ a specification. If every formula in $VCLogic(\Pi, P, Q)$ is valid, then every method of Π satisfies its contract and the triple $\{P\} S_{\text{main}} \{Q\}$ is valid.*

Verification of a program with methods, contracts, loops and recursion has thus been reduced, once again, to the validity of a finite set of formulas over indexed copies of the variables, each quantifier-free whenever the annotations are, and each discharged by a single call to a theorem prover. Nothing in the reduction inspects a callee's body, and nothing unrolls a loop.

6.8 An example

```

method Abs (x) returns (y)
  requires T
  ensures y ≥ 0 ∧ (y = old(x) ∨ y = -old(x))
  { if (x < 0) then { y := 0 - x } else { y := x }; skip }

method Dist (p, q) returns (d)
  requires T
  ensures d ≥ 0
  { t := p - q; (d) := Abs(t); skip }

```

Writing Q_{Abs} for $y \geq 0 \wedge (y = x \vee y = -x)$ — recall that $\text{old}(x)$ is just x , since Abs cannot assign to its in-parameter — the blocks for Abs are the two paths through its conditional:

$$\begin{aligned} \{T\} \quad \text{assume } x < 0 \cdot y := 0 - x \quad \{Q_{\text{Abs}}\} \\ \{T\} \quad \text{assume } \neg(x < 0) \cdot y := x \quad \{Q_{\text{Abs}}\} \end{aligned}$$

The call $(d) := \text{Abs}(t)$ is well-formed: its out-target d does not occur in its argument t , so no temporary is needed. The blocks for Dist are

$$\begin{aligned} &(\text{the precondition of Abs}) \\ &\{T\} \quad t := p - q \quad \{T\} \\ &(\text{the continuation}) \\ &\{T\} \quad t := p - q \cdot \text{havoc } d \cdot \text{assume } (d \geq 0 \wedge (d = t \vee d = -t)) \quad \{d \geq 0\} \end{aligned}$$

The second becomes, after Section 5.4, the quantifier-free formula

$$\underbrace{t_1 = p_0 - q_0 \wedge \dots}_{t := p - q} \wedge \underbrace{\bigwedge_{y \neq d} (y_2 = y_1)}_{\text{havoc } d} \wedge (d_2 \geq 0 \wedge (d_2 = t_2 \vee d_2 = -t_2)) \Longrightarrow d_2 \geq 0$$

which is valid, and trivially so — note that d_2 appears in no defining equation, which is the encoding's way of saying that the only thing known about the returned value is what Abs 's contract asserts, and that $t_2 = t_1$ comes from the frame conjuncts of the havoc, which is what makes reading t after the havoc legitimate. The body of Abs plays no part in this formula; only its contract does.

Had we written $(t) := Abs(t)$ instead, the call would be ill-formed, and Lemma 6.1 would direct us to write $a := t; (t) := Abs(a)$.

6.9 A recursive example: Fibonacci

Nothing so far has exercised recursion, which is the case the whole modular scheme was built for. Consider

```
method Fib (n) returns (r)
  requires n ≥ 0
  ensures r ≥ 1
  { if (n < 2) then { r := 1 }
    else { (a) := Fib(n - 1); (b) := Fib(n - 2); r := a + b }; skip }
```

with $X_{Fib} = \{n, r, a, b\}$. The method computes the Fibonacci numbers in the convention $(0) = (1) = 1$, but the contract does not say so: it claims only that the result is positive. A deliberately weak specification is the right thing here, because it lets the example be about the recursion rather than about arithmetic. We return to the cost of strengthening it at the end.

The blocks. Applying Blk to the conditional splits the path in two; the else-branch then meets two calls in succession, each contributing one closed block and extending the path. Writing

$$\begin{aligned} C_1 &= \text{assume } \neg(n < 2) \cdot \text{havoc } a \cdot \text{assume } (a \geq 1) \\ C_2 &= C_1 \cdot \text{havoc } b \cdot \text{assume } (b \geq 1) \end{aligned}$$

the four blocks of $Blk(n \geq 0, \epsilon, S_{Fib}, r \geq 1)$ are

$$\begin{aligned} B_1 &: \{n \geq 0\} \text{ assume } n < 2 \cdot r := 1 \{r \geq 1\} \\ B_2 &: \{n \geq 0\} \text{ assume } \neg(n < 2) \{n - 1 \geq 0\} \\ B_3 &: \{n \geq 0\} C_1 \{n - 2 \geq 0\} \\ B_4 &: \{n \geq 0\} C_2 \cdot r := a + b \{r \geq 1\} \end{aligned}$$

B_2 and B_3 are the two precondition obligations, one per call site, each closed at its call and therefore checked in the state just before it. B_4 carries both assumed postconditions, because a call is not a cut point: the knowledge gained from the first call is still available at the second and at the end.

The logic. B_4 is the interesting one. Its path has three fuel-consuming atoms — two havocs and one assignment — so $len = 3$, while the assume statements sit at whatever index has been reached. Writing the frame conjuncts out,

$$\begin{array}{ll} n_0 \geq 0 & (\text{precondition, at index 0}) \\ \wedge \neg(n_0 < 2) & (\text{the else branch}) \\ \wedge n_1 = n_0 \wedge r_1 = r_0 \wedge b_1 = b_0 & (\text{havoc } a; \text{ note } a_1 \text{ is unconstrained}) \\ \wedge a_1 \geq 1 & (\text{contract of the first call}) \\ \wedge n_2 = n_1 \wedge r_2 = r_1 \wedge a_2 = a_1 & (\text{havoc } b) \\ \wedge b_2 \geq 1 & (\text{contract of the second call}) \\ \wedge r_3 = a_2 + b_2 \wedge n_3 = n_2 \wedge a_3 = a_2 \wedge b_3 = b_2 & (r := a + b) \\ \implies r_3 \geq 1 & \end{array}$$

The frame conjuncts identify $a_1 = a_2 = a_3$ and $b_2 = b_3$, so the formula says: if $a \geq 1$ and $b \geq 1$ then $a + b \geq 1$. The other three are equally small: B_1 reduces to $1 \geq 1$, and B_2, B_3 to “ $n \geq 2$ implies $n - 1 \geq 0$ ” and “ $n \geq 2$ implies $n - 2 \geq 0$ ”. All four are quantifier-free formulas of linear integer arithmetic, decided instantly using an SMT solver.

Two features of this are worth pausing on. First, B_4 never uses $n_0 \geq 0$ or $\neg(n_0 < 2)$: the postcondition follows from the two contracts alone. The precondition $n \geq 0$ is in the contract not to help prove $r \geq 1$, but to keep the recursion inside its own domain — it is what B_2 and B_3 discharge. Second, the numbers a and b enter the formula only through $a_1 \geq 1$ and $b_2 \geq 1$; they have no defining equations, because a havocked variable has none. The encoding thereby says, quite literally, that all the caller knows about a returned value is what the callee’s contract asserts.

What the example shows.

- **The body of Fib occurs in no verification condition about a call to Fib .** The recursive calls are discharged entirely by the contract; the recursion is never unrolled, and the four blocks above would be unchanged if the body were replaced by any other implementation meeting the same contract.
- **A method is verified assuming its own contract**, and this is not circular: Theorem 6.3 inducts on the height of the derivation of $\langle S_{Fib}, \iota \rangle \Downarrow \tau$, and the premise supplied by rule (CALL) for $Fib(n - 1)$ is a strictly shorter derivation than the one it belongs to. Nothing in the argument mentions the argument $n - 1$ being smaller than n .
- **And therefore nothing here proves termination.** The implementation

`method  $Fib$  ( $n$ ) returns ( $r$ ) requires  $n \geq 0$  ensures  $r \geq 1$  { ( $r$ ) :=  $Fib(n)$ ; skip }`

generates the two blocks $\{n \geq 0\} \epsilon \{n \geq 0\}$ and $\{n \geq 0\} \text{ havoc } r \cdot \text{ assume } (r \geq 1) \{r \geq 1\}$, both trivially valid. It satisfies its contract in the sense of Section 6.3, vacuously, because it has no terminating executions at all. Proving termination needs a ranking function that strictly decreases at each call — here n itself, bounded below by the precondition — and a corresponding extra verification condition at every call site. We do not treat it.