

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

Madhusudan Parthasarathy

October 22, 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 $y \in X$. 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 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.

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.

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

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. $\blacksquare$

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.

7 Programs with a Heap

Every program so far has manipulated a fixed, finite set of scalar variables. We now add a *heap*: an unbounded, dynamically allocated collection of locations with pointer and data fields. This is where verification becomes genuinely hard, and where the encoding of Section 5.4 has to be extended with some care, because two things that were free before are no longer free: saying what a piece of code does *not* change, and saying what a called method might have done to the heap.

Our treatment follows the modelling used in the Boogie-based verification of Murali, Rivera and Madhusudan, *Predictable Verification using Intrinsic Definitions* (PLDI 2024). Three choices from there drive everything below.

- Each field f becomes a *map* (array) H_f indexed by locations, and a field update becomes an array update. There is no single monolithic “heap” object.
- Allocation is modelled by a ghost set A of allocated locations which only ever grows.
- The effect of a method call on the heap is described by a *parameterized map update*: a macro, built from an if-then-else, that says “at locations the call was allowed to modify, the new heap is some fresh uninterpreted map; everywhere else it is the old heap”. Crucially this is a *definition* of the new map, not a universally quantified axiom about it, and this is what keeps the verification conditions quantifier-free.

7.1 States

Fix a sort Loc of locations together with a distinguished constant $\mathbf{nil} : Loc$. Fix also a finite set of *field names*

$$F = F_p \uplus F_d$$

where each $f \in F_p$ is a *pointer field*, of type $Loc \rightarrow Loc$, and each $f \in F_d$ is a *data field*, of type $Loc \rightarrow \mathbb{Z}$. The set F is fixed for the whole program: there is a single heap and a single, fixed collection of field names.

Program variables are now of two kinds, $X = X_d \uplus X_p$: the *data variables*, ranging over $\mathbb{Z}$ as before, and the *pointer variables*, ranging over Loc .

A *state* is a triple $\theta = (\sigma, A, H)$ where

- σ is a store, mapping X_d to $\mathbb{Z}$ and X_p to Loc ;
- $A \subseteq Loc$ is a finite set of *allocated* locations with $\mathbf{nil} \notin A$;
- $H = (H_f)_{f \in F}$ assigns to each field name a total map of the appropriate type.

Only the values of H_f on A are meaningful; making the maps total is a modelling convenience that lets us treat them as arrays.

A state is *well formed* if pointers never dangle:

$$\sigma(x) \in A \cup \{\mathbf{nil}\} \text{ for every } x \in X_p, \quad H_f(\ell) \in A \cup \{\mathbf{nil}\} \text{ for every } f \in F_p, \ell \in A.$$

We will see in Section 7.3 that the semantics preserves well-formedness, and in Section 7.6 how the verification conditions are allowed to use this.

7.2 Syntax

Expressions stay *pure*: no expression dereferences the heap. This is the usual three-address normal form, and it costs nothing, since $x := y.f.g$ can always be written $t := y.f; x := t.g$. So $AExp$ is as in Section 1 but over X_d only, and we add location expressions

$$LExp ::= -x \mid \mathbf{nil} \quad (x \in X_p)$$

and extend $BExp$ with equalities $LExp = LExp$ between location expressions.

The statement grammar of Section 6 gains three productions:

$$S ::= \dots \mid x := \mathbf{new} \mid x := y.f \mid y.f := e$$

where in $x := y.f$ we require $y \in X_p$, and $x \in X_p$ if $f \in F_p$ while $x \in X_d$ if $f \in F_d$; symmetrically for $y.f := e$, where e is an $LExp$ or an $AExp$ according to the kind of f . There is no **free** statement: storage is reclaimed by garbage collection, which is invisible to the semantics and to the proof system.

Contracts with a modifies clause. A method declaration now has the form

```
method  $m(x_1, \dots, x_k)$  returns  $(y_1, \dots, y_l)$ 
  requires  $P_m$ 
  modifies  $M_m$ 
  ensures  $Q_m$ 
  {  $S_m$  }
```

where, writing $\vec{x}^m, \vec{y}^m$ for the in- and out-parameters as before,

- P_m is a formula over $\vec{x}^m$, A and H ;

- M_m is a *set-valued term* over $\vec{x}^m$, A and H , denoting a subset of Loc . We do not fix a syntax for it. All we require is that membership $\ell \in M_m$ is expressible as a quantifier-free formula with ℓ free — for instance $\{p, q\}$, or $\{\ell : H_{next}[\ell] = p\}$, or the value of a set-valued ghost field. Everything below depends on the set only through that membership test.
- Q_m is a formula over $\vec{x}^m$, $\vec{y}^m$, the post-state A and H , and the pre-state, written with *old*: $old(A)$ and $old(H_f)$, and more generally $old(e)$ for an expression e evaluated in the pre-state.

In-parameters remain immutable, as in Section 6.1; the well-formedness conditions listed there carry over verbatim, including the requirement that no out-target of a call occurs in any of its arguments.

old stops being vacuous. Section 6.1 remarked that *old* could be erased there, because the in-parameters were the only pre-state a postcondition could mention and they never changed; and that it “stops being vacuous the moment the language acquires state that survives a call”. That moment has arrived. The heap and the allocation set are shared between caller and callee and are mutated by the callee, so a postcondition such as

$$H_{val}[p] = old(H_{val}[p]) + 1$$

is not expressible without it. From Section 7.6 onwards, $old(H_f)$ and $old(A)$ are realised by *frozen* copies $\overline{H_f}$ and $\overline{A}$, which are ordinary read-only variables of the verification problem and never appear in program text — exactly the device Section 6.1 described and then did not need.

7.3 Semantics as a proof system

Judgements now take the form $\langle S, \theta \rangle \Downarrow \theta'$ with θ, θ' states, and we add a distinguished *error* state $\perp$ reached on a null dereference. The rules of Section 6.2 carry over with σ replaced by θ throughout, reading and writing only the σ component; we give the new rules.

$$\begin{array}{c}
\frac{\ell \notin A \cup \{\mathbf{nil}\} \quad A' = A \cup \{\ell\} \quad H'_f = H_f[\ell \mapsto d_f] \text{ for each } f \in F}{\langle x := \mathbf{new}, (\sigma, A, H) \rangle \Downarrow (\sigma[x \mapsto \ell], A', H')} \text{ (NEW)} \\
\\
\frac{\sigma(y) \neq \mathbf{nil}}{\langle x := y.f, (\sigma, A, H) \rangle \Downarrow (\sigma[x \mapsto H_f(\sigma(y))], A, H)} \text{ (READ)} \quad \frac{\sigma(y) = \mathbf{nil}}{\langle x := y.f, (\sigma, A, H) \rangle \Downarrow \perp} \text{ (READ-ERR)} \\
\\
\frac{\sigma(y) \neq \mathbf{nil} \quad H'_f = H_f[\sigma(y) \mapsto eval(\theta, e)]}{\langle y.f := e, (\sigma, A, H) \rangle \Downarrow (\sigma, A, H')} \text{ (WRITE)} \quad \frac{\sigma(y) = \mathbf{nil}}{\langle y.f := e, (\sigma, A, H) \rangle \Downarrow \perp} \text{ (WRITE-ERR)}
\end{array}$$

Here d_f is the default value of field f : $\mathbf{nil}$ for $f \in F_p$ and 0 for $f \in F_d$. In (NEW) the location ℓ is arbitrary subject to being unallocated, so allocation is nondeterministic — a program may not depend on which address it receives. Errors propagate: if any premise of a rule derives $\perp$, so does the conclusion, and $\langle S, \perp \rangle \Downarrow \perp$ for every S .

Rule (CALL) of Section 6.2 is unchanged in shape, but note what it now does: the callee is entered with the caller’s A and H , and returns the callee’s A and H . The heap is global; only the store is private.

$$\frac{v_i = eval(\theta, e_i) \text{ and } \iota(x_i^m) = v_i \text{ for } 1 \leq i \leq k \quad \langle S_m, (\iota, A, H) \rangle \Downarrow (\tau, A', H')}{\langle (\vec{z}) := m(\vec{e}), (\sigma, A, H) \rangle \Downarrow (\sigma[\vec{z} \mapsto \tau(\vec{y}^m)], A', H')} \text{ (CALL)}$$

Lemma 7.1 (Well-formedness is preserved) *If θ is well formed and $\langle S, \theta \rangle \Downarrow \theta'$ with $\theta' \neq \perp$, then θ' is well formed. Moreover $A \subseteq A'$: the allocation set only grows.*

Proof. Induction on the height of the derivation. (NEW) adds ℓ to A and initialises its fields to $\mathbf{nil}$ or 0; no rule removes a location from A ; and the only rule writing a pointer field is (WRITE), whose right-hand side is an *LExp*, hence $\mathbf{nil}$ or the value of a pointer variable, which is in $A \cup \{\mathbf{nil}\}$ by hypothesis. ■

7.4 Contracts and correctness

Fix m and a well-formed state $\theta = (\sigma, A, H)$ with $\sigma \models P_m$, and let ι agree with the calling convention. The method m *satisfies its contract* if for every such entry state (ι, A, H) ,

- (i) *safety*: $\langle S_m, (\iota, A, H) \rangle \Downarrow \perp$ does not hold;
- (ii) *functional correctness*: for every (τ, A', H') with $\langle S_m, (\iota, A, H) \rangle \Downarrow (\tau, A', H')$, the postcondition Q_m holds when its unbarred symbols are read in (τ, A', H') and its *old*($\cdot$) symbols in (ι, A, H) ;
- (iii) *framing*: for every $\ell \in A$ with $\ell \notin M_m(\iota, A, H)$ and every $f \in F$, $H'_f(\ell) = H_f(\ell)$.

Condition (iii) is the meaning of the modifies clause: the callee may change fields only at locations it declared, *or* at locations that were not allocated when it started — freshly allocated locations are always free to modify. Note that M_m is evaluated in the pre-state; a modifies clause is a promise made on entry.

Clause (i) is new. Until now a stuck computation was simply not a terminating computation, and partial correctness ignored it. A null dereference must not be ignored, so the error state is explicit and safety is part of what a contract asserts.

7.5 Basic paths over heaps

To generate basic blocks we regard the heap maps and the allocation set as *state variables* alongside the program variables. That is, for the purposes of Sections 4.2 and 5.4 the variable set of method m is

$$X_m^\# = X_m \uplus \{H_f : f \in F\} \uplus \{A\} \uplus \{\overline{H_f} : f \in F\} \uplus \{\overline{A}\},$$

where the barred symbols are the frozen copies used for *old*. Each H_f has an array sort, A and $\overline{A}$ a set sort; otherwise they behave exactly like the integer variables of Section 4.2, receiving an index at each step and a frame conjunct when they do not change.

Basic paths gain three atoms beyond those of Sections 4.1 and 6.4:

$$\begin{array}{ll} x := H_f[t] & (a \text{ select: read field } f \text{ of } t) \\ H_f := H_f[t \mapsto v] & (a \text{ store: write field } f \text{ of } t) \\ A := A \cup \{t\} & (\text{record an allocation}) \end{array}$$

together with the parameterized map update that summarizes a call, defined next. All four are ordinary assignments to state variables; nothing about Section 5's *Blk*, *SP* or Theorem 5.4 changes.

The frame macro. Let h be a map term, M a set term, a a set term and g a fresh uninterpreted function symbol of the same type as h . Define

$$fr(h, M, a, g) \equiv \Lambda \ell. \text{ite}(\ell \in M \vee \ell \notin a, g(\ell), h[\ell])$$

and, for the allocation set, with p a fresh uninterpreted predicate,

$$gr(a, p) \equiv \Lambda \ell. (\ell \in a \vee p(\ell)).$$

Read fr as: “at a location the callee was allowed to touch — either because it is in the modifies set, or because it was not yet allocated — the new map takes some unconstrained value; everywhere else it agrees with the old map.” Read gr as: “the new allocation set is the old one together with whatever else the callee allocated.” Note that gr makes monotonicity of A true *by construction*, with no axiom to state.

The essential point, and the reason for writing these as macros rather than as symbols constrained by axioms, is taken up in Section 7.7: Λ is not a quantifier here but a piece of notation for a defined symbol. Every occurrence of $fr(h, M, a, g)[t]$ in a formula is to be replaced by $\text{ite}(t \in M \vee t \notin a, g(t), h[t])$.

7.6 Verification condition generation

Fix a method m under verification. Two obligations recur, so we name them. At any point of a path, write

$$\text{mayWrite}(t) \equiv t \in M_m \vee t \notin \bar{A}$$

for the assertion that m is permitted to modify location t : it is in m 's declared modifies set, or it was not allocated when m began.

The clauses of Blk from Sections 5.2 and 6.5 are extended as follows.

Field read.

$$\begin{aligned} \text{Blk}(A_0, C, (x := y.f); S, B) &= \{ \{A_0\} C \{y \neq \text{nil}\} \} && (\text{memory safety}) \\ &\cup \text{Blk}(A_0, C \cdot x := H_f[y] \cdot \text{assume } \gamma_f(x), S, B) \end{aligned}$$

where $\gamma_f(x)$ is $x \in A \vee x = \text{nil}$ when $f \in F_p$ and T otherwise. The assume is not an assumption we are entitled to make freely: it is a quantifier-free instance of the well-formedness invariant, licensed by Lemma 7.1, and it is what lets later dereferences of x be justified.

Field write.

$$\begin{aligned} \text{Blk}(A_0, C, (y.f := e); S, B) &= \{ \{A_0\} C \{y \neq \text{nil}\} \} && (\text{memory safety}) \\ &\cup \{ \{A_0\} C \{\text{mayWrite}(y)\} \} && (\text{modifies discipline}) \\ &\cup \text{Blk}(A_0, C \cdot H_f := H_f[y \mapsto e], S, B) \end{aligned}$$

Allocation. With u a fresh location variable,

$$\text{Blk}(A_0, C, (x := \text{new}); S, B) = \text{Blk}(A_0, C \cdot C_{\text{new}}, S, B)$$

$$C_{\text{new}} = \text{havoc } u \cdot \text{assume } (u \notin A \wedge u \neq \text{nil}) \cdot A := A \cup \{u\} \cdot (H_f := H_f[u \mapsto d_f])_{f \in F} \cdot x := u$$

where $(H_f := H_f[u \mapsto d_f])_{f \in F}$ abbreviates the string of one assignment per field, in any order. Allocation generates no proof obligation. It cannot fail, and by Lemma 7.1 the new location is allocated with default fields, which is exactly what the atoms say.

Call. For a call $(\bar{z}) := m'(\bar{e})$, write $P' = P_{m'}[\bar{e}/\bar{x}^{m'}]$, $M' = M_{m'}[\bar{e}/\bar{x}^{m'}]$ and $Q' = Q_{m'}[\bar{e}/\bar{x}^{m'}, \bar{z}/\bar{y}^{m'}]$, and let g_f (one per $f \in F$) and p be *fresh* uninterpreted symbols for this call site. Then

$$\begin{aligned} \text{Blk}(A_0, C, ((\bar{z}) := m'(\bar{e})); S, B) &= \{ \{A_0\} C \{P'\} \} && (\text{the call is legal}) \\ &\cup \{ \{A_0\} C \{M' \cap A \subseteq M_m \cup (A \setminus \bar{A})\} \} && (\text{the call is within budget}) \\ &\cup \text{Blk}(A_0, C \cdot C_{\text{call}}, S, B) \end{aligned}$$

where the call is summarized by the single parallel step

$$C_{\text{call}} = \text{havoc } \bar{z} \cdot \left[A := gr(A, p), \quad H_f := fr(H_f, M', A, g_f) \text{ for each } f \in F \right] \cdot \text{assume } Q'$$

in which all right-hand sides are evaluated in the state before the step, so that M' and the occurrences of A inside fr refer to the pre-call allocation set, and the *old* symbols of Q' refer to the pre-call H_f and A .

Finally, the verification conditions of a program are as in Section 6.5, with the frozen copies tied down at entry:

$$VC(\Pi, P, Q) = \text{Blk}(P^b, \epsilon, S_{\text{main}}, Q) \cup \bigcup_{m \in \Pi} \text{Blk}(P_m^b, \epsilon, S_m, Q_m), \quad P_m^b = P_m \wedge \bar{H} = H \wedge \bar{A} = A \wedge W_m$$

where $W_m = \bigwedge \{ x \in A \vee x = \text{nil} : x \in X_m \cap X_p \}$ is the well-formedness invariant instantiated at the pointer variables of m — again a quantifier-free consequence of Lemma 7.1, assumed rather than proved, and the reason a caller may conclude that a non-*nil* pointer parameter is allocated.

Four remarks.

- **The modifies clause is checked, not assumed.** A contract that declares a modifies set is worthless unless the body respects it, and the two obligations $\text{mayWrite}(y)$ at each write and $M' \cap A \subseteq M_m \cup (A \setminus \overline{A})$ at each call are what enforce it. Together they give the method's own framing obligation (iii) by construction, with no extra check at the end of the body: if every write in the body is to a location that is in M_m or was allocated after entry, then no location of $\overline{A} \setminus M_m$ is ever written.
- **Locations allocated during the call are covered.** In $\text{fr}(H_f, M', A, g_f)$ the disjunct $\ell \notin A$ uses the *pre-call* allocation set. So a location the callee allocates gets its fields from g_f , which is right: the caller knows nothing about them beyond what Q' says. Symmetrically, a location that was allocated and not in M' keeps its old value, which is the frame the caller needs.
- **A call is still not a cut point.** As in Section 6.5, the two obligations are closed at the call and the path continues. What the caller knew about the heap outside M' survives the call — not because it is re-established, but because fr is defined to agree with the old map there.
- **Everything except one obligation is quantifier-free.** The membership tests $\ell \in M'$, $\ell \notin A$ and $\text{mayWrite}(y)$ are quantifier-free formulas instantiated at a term. The one exception is the inclusion $M' \cap A \subseteq M_m \cup (A \setminus \overline{A})$, which is inherently a statement about all locations. It is confined to one obligation per call site and does not contaminate the frame reasoning; in practice one either restricts modifies clauses to a shape for which inclusion is quantifier-free (finite sets of terms, for instance, which covers most uses) or accepts a single quantified query there.

7.7 From basic blocks to logic

The blocks above are ordinary basic blocks over the variable set $X_m^\sharp$, so Section 5.4 applies unchanged: for a block $\{A_0\} C \{B\}$ with $k = \text{len}(C)$ the verification condition is

$$A_0[\vec{x}_0/\vec{x}] \wedge PF(C) \implies B[\vec{x}_k/\vec{x}].$$

What needs saying is how the new atoms contribute to PF , and why the result contains no quantifiers.

Indexing proceeds exactly as before, with H_f^i and A^i the i -th copies. The clauses are the assignment clause of Section 4.2, specialized:

$$\begin{aligned} PF(i, x := H_f[t] \cdot bp) &= (x_{i+1} = H_f^i[t_i]) \wedge \text{frame} \wedge PF(i+1, bp) \\ PF(i, H_f := H_f[t \mapsto v] \cdot bp) &= (H_f^{i+1} = H_f^i[t_i \mapsto v_i]) \wedge \text{frame} \wedge PF(i+1, bp) \\ PF(i, A := A \cup \{t\} \cdot bp) &= (A^{i+1} = A^i \cup \{t_i\}) \wedge \text{frame} \wedge PF(i+1, bp) \end{aligned}$$

and, for the call summary, the parallel step defines all of the new copies at once:

$$\begin{aligned} PF(i, [A := \text{gr}(A, p), H_f := \text{fr}(H_f, M', A, g_f)] \cdot bp) = \\ (A^{i+1} = \text{gr}(A^i, p)) \wedge \bigwedge_{f \in F} (H_f^{i+1} = \text{fr}(H_f^i, M'_i, A^i, g_f)) \wedge \text{frame} \wedge PF(i+1, bp) \end{aligned}$$

where *frame* is, as always, the conjunction of $u_{i+1} = u_i$ over the state variables u not written by the atom — which now includes the frozen copies $\overline{H_f}, \overline{A}$, written nowhere and therefore carried along unchanged, so that $\text{old}(H_f)$ may be read as $\overline{H_f}$ at any index.

Why this is quantifier-free. An equation such as $H_f^{i+1} = \text{fr}(H_f^i, M', A^i, g_f)$ looks like a statement about all locations, and if we read it as the axiom

$$\forall \ell. H_f^{i+1}[\ell] = \text{ite}(\ell \in M' \vee \ell \notin A^i, g_f(\ell), H_f^i[\ell])$$

then it is one, and the verification condition acquires a quantifier for every field at every call site. That is the naive encoding, and it is what makes heap verification depend on a solver's quantifier heuristics.

We do not read it that way. H_f^{i+1} is not a new symbol constrained by an axiom; it is an *abbreviation*, and the equation is its definition. A verification condition mentions only finitely many location terms, so we may simply expand: every occurrence of $H_f^{i+1}[t]$ in the formula is replaced by

$$ite(t \in M' \vee t \notin A^i, \quad g_f(t), \quad H_f^i[t]),$$

and likewise every occurrence of $A^{i+1}(t)$ by $A^i(t) \vee p(t)$, and every $H_f^i[t]$ arising from a store by the usual read-over-write. Expansion terminates, since each step decreases the index, and the result is a quantifier-free formula over the theories of integers, arrays and sets — all decidable, and combined decidably. Nothing in it universally quantifies over locations.

Two consequences are worth drawing out. First, the size of the expansion is bounded by (number of terms) $\times$ (number of heap versions), so a path with many call sites produces a larger but still quantifier-free formula; this is the cost of the method, and it is a cost in formula size rather than in proof difficulty. Second, this is precisely the distinction between a symbol that is *defined* and one that is *axiomatized*, and it is the reason Murali et al. report an order-of-magnitude speedup over an encoding that states the same fact with a quantifier.

Corollary 7.1.1 (Modular verification with a heap) *Let Π be a well-formed program with contracts and modifies clauses, and $\{P\}S_{main}\{Q\}$ a specification. If every formula in $VCLogic(\Pi, P, Q)$ is valid then every method of Π satisfies its contract — in particular no execution dereferences **nil**, and every method respects its modifies clause — and $\{P\}S_{main}\{Q\}$ is valid.*

The proof follows Theorem 6.3: strong induction on the height of the derivation, with the new cases for (NEW), (READ), (WRITE) and their error rules handled by the corresponding obligations, and with the call case as before except that one must now check that C_{call} over-approximates the callee. That last step is exactly the content of clause (iii) of Section 7.4: the callee guarantees $H'_f(\ell) = H_f(\ell)$ for $\ell \in A \setminus M'$, which is what $fr(H_f, M', A, g_f)$ asserts at those ℓ ; at the remaining locations fr asserts nothing, since g_f is fresh and unconstrained, so any H'_f the callee produces is a possible value of the summary.

7.8 An example: aliasing across two calls

Let $F_p = \{next\}$ and $F_d = \{val\}$, and consider

```
method Inc (p) returns ()
  requires p ≠ nil
  modifies {p}
  ensures H_val[p] = old(H_val[p]) + 1 ∧ H_next[p] = old(H_next[p])
  { t := p.val; t := t + 1; p.val := t; skip }

method IncBoth (p, q) returns ()
  requires p ≠ nil ∧ q ≠ nil ∧ p ≠ q
  modifies {p, q}
  ensures H_val[p] = old(H_val[p]) + 1 ∧ H_val[q] = old(H_val[q]) + 1
  { () := Inc(p); () := Inc(q); skip }
```

Note that *Inc* must promise $H_{next}[p]$ unchanged explicitly: a modifies clause names *locations*, not fields, so declaring $\{p\}$ licenses changing every field of p .

Blocks for *Inc*. The body has two dereferences of p and one write, giving three obligations and one final block:

```
{Pb} ∈ {p ≠ nil}
{Pb} t := H_val[p] · t := t + 1 {p ≠ nil}
{Pb} t := H_val[p] · t := t + 1 {mayWrite(p)}
{Pb} t := H_val[p] · t := t + 1 · H_val := H_val[p ↦ t] {QInc}
```

with $P^b = (p \neq \text{nil} \wedge \overline{H} = H \wedge \overline{A} = A)$. The third is $p \in \{p\} \vee p \notin \overline{A}$, which is immediate. The last expands, by read-over-write, to $H_{val}^0[p] + 1 = \overline{H}_{val}[p] + 1$ and $H_{next}^0[p] = \overline{H}_{next}[p]$, both immediate from $\overline{H} = H$ at index 0.

The interesting block: *IncBoth*. Its path is the two call summaries in sequence. Write g, g' for the fresh maps of the first and second call site, p_1, p_2 for their fresh allocation predicates, and let indices 0, 1, 2 be the states before the first call, between the calls, and after the second. The path formula contributes, among the frame conjuncts,

$$\begin{aligned} A^1 &= gr(A^0, p_1), & H_{val}^1 &= fr(H_{val}^0, \{p\}, A^0, g_{val}), \\ A^2 &= gr(A^1, p_2), & H_{val}^2 &= fr(H_{val}^1, \{q\}, A^1, g'_{val}), \end{aligned}$$

together with the two assumed postconditions

$$H_{val}^1[p] = H_{val}^0[p] + 1 \quad \text{and} \quad H_{val}^2[q] = H_{val}^1[q] + 1,$$

and the goal $H_{val}^2[p] = \overline{H}_{val}[p] + 1 \wedge H_{val}^2[q] = \overline{H}_{val}[q] + 1$, with $\overline{H} = H^0$.

Now expand the macros at the only two location terms that occur, p and q :

$$\begin{aligned} H_{val}^2[p] &= ite(p \in \{q\} \vee p \notin A^1, g'_{val}(p), H_{val}^1[p]) \\ H_{val}^1[q] &= ite(q \in \{p\} \vee q \notin A^0, g_{val}(q), H_{val}^0[q]) \end{aligned}$$

The precondition gives $p \neq q$, so both guards reduce to the allocation tests. For those, $p, q \in A^0$ comes from the conjunct $W_{IncBoth}$ of P^b — a pointer parameter is **nil** or allocated, and both are non-**nil** — and $p \in A^1$ then follows by expanding gr at p : $A^1(p) \equiv A^0(p) \vee p_1(p)$. Hence both guards are false, and the two equations collapse to $H_{val}^2[p] = H_{val}^1[p]$ and $H_{val}^1[q] = H_{val}^0[q]$, from which the goal follows by the two assumed postconditions.

That derivation is the entire content of the example, and every step of it is a propositional or arithmetic consequence of instantiating the *ite* at p and at q . Two things are worth noticing. The disequality $p \neq q$ is doing all the work: drop it from the precondition and the second call's summary no longer preserves the first call's effect, and the verification condition becomes invalid — correctly, since *IncBoth*(p, p) increments once by one and once by one again, but the specification as written claims two independent increments of two distinct locations. And monotonicity of A was needed, in the form $p \in A^0 \Rightarrow p \in A^1$; because gr builds monotonicity into the definition, this too was a matter of expanding a macro rather than appealing to an axiom.

8 Hoare Logic

Everything so far has been *computational*: given a program and its annotations, a procedure produces a finite set of logical formulas, and the program is correct if they are all valid. Hoare logic takes the other route. It is a *proof system* — a finite set of axioms and inference rules for deriving Hoare triples — in which correctness is established by exhibiting a derivation rather than by running an algorithm.

The two are closely related, and the relationship is worth stating up front. A derivation in the proof system below has exactly one place where the prover must invent something: the invariant in the while rule, and (in Section 8.3) the contract of each method. Fix those and the rest of the derivation is forced, and its leaves are precisely the implications collected by *VC* in Sections 5 and 6. Verification-condition generation is Hoare logic with the creative steps supplied in advance and the bookkeeping done mechanically.

Throughout this section we work with *unannotated* programs: the proof system does not read an **Inv** annotation, because the while rule leaves the invariant for the prover to choose. Section 5's annotation is nothing other than that choice, written down in the program text.

8.1 The proof system for while programs

Judgements have the form $\vdash \{P\} S \{Q\}$, read “the triple $\{P\} S \{Q\}$ is derivable”. Validity, $\models \{P\} S \{Q\}$, remains as defined in Section 3.

$$\begin{array}{c}
\frac{}{\vdash \{P\} \text{ skip } \{P\}} \text{ (SKIP)} \quad \frac{}{\vdash \{Q[e/x]\} x := e \{Q\}} \text{ (ASSIGN)} \quad \frac{}{\vdash \{P\} \text{ assume } b \{P \wedge b\}} \text{ (ASSUME)} \\
\\
\frac{\vdash \{P\} S_1 \{R\} \quad \vdash \{R\} S_2 \{Q\}}{\vdash \{P\} S_1; S_2 \{Q\}} \text{ (SEQ)} \\
\\
\frac{\vdash \{P \wedge b\} S_1 \{Q\} \quad \vdash \{P \wedge \neg b\} S_2 \{Q\}}{\vdash \{P\} \text{ if } b \text{ then } \{S_1\} \text{ else } \{S_2\} \{Q\}} \text{ (IF)} \\
\\
\frac{\vdash \{I \wedge b\} S \{I\}}{\vdash \{I\} \text{ while } (b) \{S\} \{I \wedge \neg b\}} \text{ (WHILE)} \\
\\
\frac{\models P \rightarrow P' \quad \vdash \{P'\} S \{Q'\} \quad \models Q' \rightarrow Q}{\vdash \{P\} S \{Q\}} \text{ (CONSEQ)}
\end{array}$$

Four observations on the rules.

- **(Assign) runs backwards.** The axiom is stated with the substituted postcondition as precondition, not with a “ x gets e ” postcondition. This is Floyd’s observation inverted, and it is what makes the rule usable: to prove $\{P\} x := e \{Q\}$ one checks $P \rightarrow Q[e/x]$ by (CONSEQ). The forward reading, with $\{P\} x := e \{ \exists v. P[v/x] \wedge x = e[v/x] \}$, is also sound but introduces a quantifier at every assignment — which is exactly why Section 5.4 computed strongest postconditions with indexed copies instead.
- **The side conditions of (Conseq) are semantic.** The premises $\models P \rightarrow P'$ and $\models Q' \rightarrow Q$ are statements about the assertion logic, not about programs, and the proof system assumes an oracle for them. This is the standard separation of concerns: Hoare logic reduces reasoning about *programs* to reasoning about *formulas*, and the latter is somebody else’s problem — in practice, an SMT solver’s.
- **(While) is the only rule that is not syntax-directed in its conclusion.** Every other rule, read bottom-up, is determined by the shape of S ; the while rule requires inventing I , which no amount of bookkeeping supplies.
- **Partial correctness only.** $\vdash \{T\} \text{ while } (\text{true}) \{ \text{skip} \} \{F\}$ is derivable, taking $I = T$. As in Sections 5 and 6, nothing here proves termination.

Derived rules. Three rules are frequently convenient and are derivable from the above, the first two immediately by (CONSEQ):

$$\begin{array}{c}
\frac{\vdash \{P_1\} S \{Q_1\} \quad \vdash \{P_2\} S \{Q_2\}}{\vdash \{P_1 \wedge P_2\} S \{Q_1 \wedge Q_2\}} \text{ (CONJ)} \quad \frac{\vdash \{P_1\} S \{Q\} \quad \vdash \{P_2\} S \{Q\}}{\vdash \{P_1 \vee P_2\} S \{Q\}} \text{ (DISJ)} \\
\\
\frac{\vdash \{P\} S \{Q\}}{\vdash \{P \wedge R\} S \{Q \wedge R\}} \text{ (INV)}
\end{array}$$

where (INV), the *rule of invariance* or *constancy*, carries the side condition $\text{free}(R) \cap \text{asgn}(S) = \emptyset$, with asgn as in Section 6.3. It says that an assertion about variables the statement does not write survives the statement. It is derivable for the language of this subsection by induction on S , and it becomes indispensable in Section 8.3, where it is what lets a caller keep what it knew across a call.

Theorem 8.1 (Soundness) *If $\vdash \{P\} S \{Q\}$ then $\models \{P\} S \{Q\}$.*

Proof. By induction on the derivation. (SKIP), (ASSIGN) and (ASSUME) are immediate from the semantics; (SEQ), (IF) and (CONSEQ) follow at once from the induction hypotheses.

For (WHILE), let $\sigma \models I$ and suppose $\langle \text{while } (b)\{S\}, \sigma \rangle \Downarrow \sigma'$. Argue by an inner induction on the height of that derivation. If it ends in (WHILE-F) then $\sigma' = \sigma$ and $\text{eval}(\sigma, b) = F$, so $\sigma' \models I \wedge \neg b$. If it ends in (WHILE-T) then $\text{eval}(\sigma, b) = T$, and there is σ_1 with $\langle S, \sigma \rangle \Downarrow \sigma_1$ and $\langle \text{while } (b)\{S\}, \sigma_1 \rangle \Downarrow \sigma'$, both of smaller height. Since $\sigma \models I \wedge b$, the outer induction hypothesis applied to the premise $\vdash \{I \wedge b\}S\{I\}$ gives $\sigma_1 \models I$, and the inner induction hypothesis then gives $\sigma' \models I \wedge \neg b$. ■

The converse — that every valid triple is derivable — holds only relative to the assertion logic, and requires that logic to be *expressive* enough to define the weakest precondition of every statement and postcondition. This is Cook's theorem, and we do not pursue it here; note only that the two premises of (CONSEQ) are where any such result must put all of its work.

8.2 Floyd annotations are Hoare derivations

The connection with Section 5 can be made precise. Recall *Blk* and *VC* from Section 5.2, defined on programs whose loops carry invariant annotations.

Theorem 8.2 *Let S be an annotated program and P, Q assertions. If every basic block in $VC(P, S, Q)$ is a valid Hoare triple, then $\vdash \{P\} \text{er}(S) \{Q\}$ is derivable.*

Proof sketch. Induction on S , following the clauses of *Blk*. The annotations supply every invariant needed by (WHILE); the straight-line segments between cut points are discharged by (ASSIGN), (ASSUME), (SEQ) and (IF); and each block $\{A\}C\{B\}$ of *VC*, being valid, provides exactly one instance of (CONSEQ). In particular the block emitted at a loop header, $\{A\}C\{I\}$, is the implication needed to enter the loop, and the body blocks $\text{Blk}(I \wedge b, \epsilon, S_1, I)$ are the premise of (WHILE). ■

Conversely, a derivation determines annotations: read off the I used at each application of (WHILE) and write it into the program. So the two presentations carry the same information, and the choice between them is a choice about who does the bookkeeping. The proof system is the better object for proving things *about* verification — as here, in Theorem 8.1 — while the verification conditions are the better object for a machine to check.

8.3 Programs with methods and calls

We now extend the system to the language of Section 6: methods with contracts, immutable in-parameters, call-by-value, multiple return values. The essential new ingredient is that derivations are relative to a set of *assumed* contracts.

Let Γ be a set of triples (P_m, m, Q_m) , one for some of the methods of Π , written as a context. Judgements become

$$\Gamma \vdash \{P\} S \{Q\}$$

and all the rules of Section 8.1 are repeated with Γ carried unchanged through every premise. The two new rules are the following.

$$\frac{(P_m, m, Q_m) \in \Gamma}{\Gamma \vdash \{P_m[\vec{e}/\vec{x}^m]\} (\vec{z}) := m(\vec{e}) \{Q_m[\vec{e}/\vec{x}^m, \vec{z}/\vec{y}^m]\}} \text{ (CALL)}$$

side condition: no z_j occurs in any e_i

$$\frac{\Gamma_\Pi \vdash \{P_m\} S_m \{Q_m\} \text{ for every method } m \text{ of } \Pi}{\text{every method of } \Pi \text{ satisfies its contract}} \text{ (REC)}$$

where $\Gamma_\Pi = \{(P_m, m, Q_m) : m \text{ a method of } \Pi\}$ is the set of *all* contracts of Π

Rule (CALL) is the proof-theoretic form of Section 6.5's *Blk* clause: substitute actuals for formals in the precondition, and actuals and targets in the postcondition. The side condition is the well-formedness requirement of Section 6.1, and it is needed here for the same reason: the conclusion's postcondition mentions $\vec{e}$, and the statement writes $\vec{z}$, so the two must be disjoint for $\vec{e}$ to still denote what it denoted before the call.

Rule (REC) is the rule for recursion, in its simultaneous form: to establish all the contracts, prove each body while assuming *all* of them, including the one being proved. Its conclusion is not a Hoare triple but a statement about the whole program, which is why it is written as a rule of a different shape; one may read it as the discharge of the hypotheses in Γ_Π .

Why (Inv) is now essential. On its own, (CALL) is far too weak. It says what the call establishes about $\vec{z}$ and nothing about anything else, so after a call the caller would have forgotten every fact it had. The invariance rule repairs this: since $asgn((\vec{z}) := m(\vec{e})) = \vec{z}$, any R not mentioning $\vec{z}$ passes through. Thus from

$$\Gamma \vdash \{P_m[\vec{e}/\vec{x}^m]\} (\vec{z}) := m(\vec{e}) \{Q_m[\dots]\}$$

one obtains, by (INV) and then (CONSEQ), the usable form

$$\Gamma \vdash \{A\} (\vec{z}) := m(\vec{e}) \{Q_m[\dots] \wedge R\} \quad \text{whenever } \models A \rightarrow P_m[\vec{e}/\vec{x}^m] \wedge R \text{ and } free(R) \cap \vec{z} = \emptyset.$$

This is the proof-theoretic counterpart of the remark in Section 6.5 that a call is not a cut point: the path carries on, and what the caller knew about its other variables is still available.

It is worth noting what we do *not* need. Hoare logic for procedures classically requires a *rule of adaptation* to relate the state at the call site to the contract, typically via auxiliary (logical) variables quantified in the contract. We escape this because of two decisions made in Section 6: in-parameters are immutable, so an actual argument e_i denotes the same value before and after the call; and Q_m may mention only $\vec{x}^m$ and $\vec{y}^m$. Together these make plain substitution sufficient. The discipline that simplified the verification conditions simplifies the proof theory in the same way and for the same reason.

Theorem 8.3 (Soundness with calls) *Let Π be a well-formed program. If $\Gamma_\Pi \vdash \{P_m\} S_m \{Q_m\}$ for every method m of Π , then every method of Π satisfies its contract. Moreover, if in addition $\Gamma_\Pi \vdash \{P\} S_{main} \{Q\}$, then $\models \{P\} S_{main} \{Q\}$.*

Proof. The statement to prove by strong induction on $h \in \mathbb{N}$ is:

Claim(h). For every judgement $\Gamma_\Pi \vdash \{P\} S \{Q\}$ that is derivable, every $\sigma \models P$ and every derivation of $\langle S, \sigma \rangle \Downarrow \sigma'$ of height at most h , we have $\sigma' \models Q$.

Assume Claim(h') for all $h' < h$ and argue by a secondary induction on the derivation of $\Gamma_\Pi \vdash \{P\} S \{Q\}$. All the rules of Section 8.1 are handled as in Theorem 8.1, noting that in each the operational premises have height less than h . (INV) uses Lemma 6.2. The one new case is (CALL).

Let S be $(\vec{z}) := m'(\vec{e})$ with $(P_{m'}, m', Q_{m'}) \in \Gamma_\Pi$, let $\sigma \models P_{m'}[\vec{e}/\vec{x}^{m'}]$, and consider a derivation of $\langle S, \sigma \rangle \Downarrow \sigma'$ of height at most h . It ends in rule (CALL) of Section 6.2, so there are $v_i = eval(\sigma, e_i)$, 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 , with $\sigma' = \sigma[\vec{z} \mapsto \tau(\vec{y}^{m'})]$.

Since $P_{m'}$ mentions only $\vec{x}^{m'}$ and $\iota(\vec{x}^{m'}) = \vec{v}$, we get $\iota \models P_{m'}$. By hypothesis $\Gamma_\Pi \vdash \{P_{m'}\} S_{m'} \{Q_{m'}\}$ is derivable, and the callee's derivation has height less than h , so Claim of that smaller height gives $\tau \models Q_{m'}$. Finally, $Q_{m'}$ mentions only $\vec{x}^{m'}$ and $\vec{y}^{m'}$; by Lemma 6.2 the in-parameters are unchanged, so $\tau(\vec{x}^{m'}) = \vec{v}$, and by the side condition no z_j occurs in any e_i , so $eval(\sigma', e_i) = v_i$. Hence $\sigma' \models Q_{m'}[\vec{e}/\vec{x}^{m'}, \vec{z}/\vec{y}^{m'}]$. ■

The apparent circularity of (REC) — each body is proved assuming the very contract it is establishing — is discharged by the induction on h , precisely as in Theorem 6.3. A recursive call appears in the proof only through Γ_Π , and when the claim is applied to it the callee's *derivation* is strictly shorter even though the program text is the same. Nothing in the argument refers to any measure on the arguments, which is why (REC) proves partial correctness and nothing more: the program $\{(\vec{r}) := m(\vec{x}); \text{skip}\}$ satisfies every contract one cares to write for m .

8.4 Weakest liberal preconditions and strongest postconditions

Two constructions have appeared repeatedly: the backward one behind (ASSIGN), and the forward one, SP , of Section 5.4. They are not independent. This subsection makes their relationship precise.

Work semantically, identifying an assertion with the set of stores satisfying it and writing $\Rightarrow$ for inclusion. For a statement S of a method m and assertions P, Q over X_m , define

$$\begin{aligned} wlp(S, Q) &= \{ \sigma \mid \text{for every } \sigma' \text{ with } \langle S, \sigma \rangle \Downarrow \sigma', \sigma' \models Q \} \\ sp(P, S) &= \{ \sigma' \mid \text{for some } \sigma \models P, \langle S, \sigma \rangle \Downarrow \sigma' \} \end{aligned}$$

the *weakest liberal precondition* of Q and the *strongest postcondition* of P . The word *liberal* marks partial correctness: wlp demands nothing of executions that fail to terminate, and indeed $wlp(S, F)$ is exactly the set of stores from which S has no terminating execution at all. The total-correctness variant is $wp(S, Q) = wlp(S, Q) \cap \{ \sigma : S \text{ terminates from } \sigma \}$, which these notes do not treat.

Lemma 8.4 *For all P, Q, S the following are equivalent:*

$$(i) \models \{P\} S \{Q\} \quad (ii) P \Rightarrow wlp(S, Q) \quad (iii) sp(P, S) \Rightarrow Q.$$

In particular $wlp(S, Q)$ is the weakest P making the triple valid, and $sp(P, S)$ the strongest such Q — whence the names.

Proof. Unfolding the definitions, each of the three says

$$\text{for all } \sigma, \sigma' : \sigma \models P \text{ and } \langle S, \sigma \rangle \Downarrow \sigma' \text{ implies } \sigma' \models Q,$$

in (ii) by grouping the quantifiers as $\forall \sigma. (\sigma \models P \rightarrow \forall \sigma'. \dots)$ and in (iii) as $\forall \sigma'. ((\exists \sigma. \dots) \rightarrow \sigma' \models Q)$. The two claims about weakest and strongest follow by taking $P := wlp(S, Q)$ in (ii) and $Q := sp(P, S)$ in (iii), each of which holds by reflexivity of $\Rightarrow$. ■

The content of the lemma is best stated as follows.

Theorem 8.5 (sp and wlp are adjoint) *For every statement S , the maps $sp(-, S)$ and $wlp(S, -)$ form a Galois connection between assertions ordered by $\Rightarrow$:*

$$sp(P, S) \Rightarrow Q \quad \text{if and only if} \quad P \Rightarrow wlp(S, Q).$$

$sp(-, S)$ is the lower (left) adjoint and $wlp(S, -)$ the upper (right) adjoint.

Proof. Immediate from Lemma 8.4: both sides are equivalent to $\models \{P\} S \{Q\}$. ■

Everything else follows formally. We spell the consequences out, since each has a concrete reading for verification.

Corollary 8.5.1 *For every S and all P, Q, P_i, Q_i :*

- (a) (unit and counit) $P \Rightarrow wlp(S, sp(P, S))$ and $sp(S, wlp(S, Q)) \Rightarrow Q$, writing $sp(S, R)$ for $sp(R, S)$;
- (b) (monotonicity) $P_1 \Rightarrow P_2$ implies $sp(P_1, S) \Rightarrow sp(P_2, S)$, and $Q_1 \Rightarrow Q_2$ implies $wlp(S, Q_1) \Rightarrow wlp(S, Q_2)$;
- (c) (triangle identities) $sp(wlp(S, sp(P, S)), S) = sp(P, S)$ and $wlp(S, sp(wlp(S, Q), S)) = wlp(S, Q)$;
- (d) (limits) wlp preserves arbitrary conjunctions and sp arbitrary disjunctions:

$$wlp(S, \bigwedge_i Q_i) = \bigwedge_i wlp(S, Q_i) \quad \text{and} \quad sp(\bigvee_i P_i, S) = \bigvee_i sp(P_i, S).$$

Proof. (a) Instantiate Theorem 8.5 at $Q := sp(P, S)$ and read it right to left, using $sp(P, S) \Rightarrow sp(P, S)$; dually at $P := wlp(S, Q)$.

(b) Suppose $P_1 \Rightarrow P_2$. By (a), $P_2 \Rightarrow wlp(S, sp(P_2, S))$, so $P_1 \Rightarrow wlp(S, sp(P_2, S))$, and Theorem 8.5 gives $sp(P_1, S) \Rightarrow sp(P_2, S)$. The other half is dual.

(c) Write $A = sp(P, S)$. By (a) $P \Rightarrow wlp(S, A)$, so by (b) $A \Rightarrow sp(wlp(S, A), S)$; and by the counit in (a) applied with $Q := A$, $sp(wlp(S, A), S) \Rightarrow A$. The second identity is dual.

(d) A right adjoint preserves all existing meets. Directly: $P \Rightarrow wlp(S, \bigwedge_i Q_i)$ iff $sp(P, S) \Rightarrow \bigwedge_i Q_i$ iff $sp(P, S) \Rightarrow Q_i$ for every i iff $P \Rightarrow wlp(S, Q_i)$ for every i iff $P \Rightarrow \bigwedge_i wlp(S, Q_i)$; now take P to be each side in turn. The claim for sp is dual. ■

Two warnings about what the corollary does *not* say. The inclusions in (a) are usually strict: $wlp(S, sp(P, S))$ is in general strictly weaker than P , because it also contains stores from which S cannot run at all, and $sp(wlp(S, Q), S)$ is in general strictly stronger than Q , because it contains only reachable stores. And the dual halves of (d) fail: wlp need not preserve disjunction. With $S = \text{havoc } x$, $Q_1 = (x \geq 0)$ and $Q_2 = (x < 0)$ we have $wlp(S, Q_1 \vee Q_2) = T$ while $wlp(S, Q_1) = wlp(S, Q_2) = F$. If S is deterministic, disjunction is preserved: at most one σ' exists, and it satisfies Q_1 or it satisfies Q_2 .

Computing them. Both admit syntactic definitions, and the asymmetry between the two is exactly the asymmetry between Sections 5.4 and the (ASSIGN) axiom:

$$\begin{aligned}
wlp(\text{skip}, Q) &= Q \\
wlp(x := e, Q) &= Q[e/x] \\
wlp(\text{assume } b, Q) &= b \rightarrow Q \\
wlp(S_1; S_2, Q) &= wlp(S_1, wlp(S_2, Q)) \\
wlp(\text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\}, Q) &= (b \rightarrow wlp(S_1, Q)) \wedge (\neg b \rightarrow wlp(S_2, Q)) \\
\\
sp(P, \text{skip}) &= P \\
sp(P, x := e) &= \exists v. P[v/x] \wedge x = e[v/x] \\
sp(P, \text{assume } b) &= P \wedge b \\
sp(P, S_1; S_2) &= sp(sp(P, S_1), S_2) \\
sp(P, \text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\}) &= sp(P \wedge b, S_1) \vee sp(P \wedge \neg b, S_2)
\end{aligned}$$

Assignment is where they differ. Going backwards it is a substitution, and the formula stays quantifier-free — but the conditional duplicates Q across both branches, so nested conditionals multiply the formula's size. Going forwards it introduces an existential at every assignment. Section 5.4 takes the forward route and defuses the quantifiers by naming the intermediate values rather than quantifying them away: $SP(A, C)$ of Section 5.4 is precisely $sp(A, C)$ for a basic path C , written with indexed copies, and the step called “discharging the check without eliminating the quantifiers” there is nothing but reading Theorem 8.5 from left to right — it replaces $sp(A, C) \Rightarrow B$ by a formula in which the existentials, now sitting in an antecedent, may simply be dropped.

Loops. Neither clause list covers the while statement, and neither can be completed by finite syntactic recursion. The remaining clauses are fixpoints, and of opposite kinds:

$$\begin{aligned}
wlp(\text{while } (b)\{S\}, Q) &= \nu X. ((b \rightarrow wlp(S, X)) \wedge (\neg b \rightarrow Q)) \\
sp(P, \text{while } (b)\{S\}) &= (\mu X. P \vee sp(X \wedge b, S)) \wedge \neg b
\end{aligned}$$

the greatest and the least fixpoint respectively. (The greatest fixpoint is the right one for wlp because partial correctness is liberal: a non-terminating execution must not be excluded.) This is the precise sense in which loops are the hard part: an assertion logic closed under these operators would make verification a matter of calculation, and Cook's expressiveness hypothesis is exactly the demand that it be. Lacking that, one supplies an invariant — which, by Lemma 8.4, is any I sandwiched as

$$P \Rightarrow I, \quad I \wedge b \Rightarrow wlp(S, I), \quad I \wedge \neg b \Rightarrow Q,$$

the three conditions that the rule (WHILE) with (CONSEQ) requires, and the three blocks that Section 5.2's *Blk* emits at a loop. The fixpoints above are the extreme choices: $wlp(\text{while } (b)\{S\}, Q)$ is the weakest invariant that works for Q , and sp with the loop's exit condition removed is the strongest that works for P . Every usable invariant lies between them.

8.5 A worked example

We give a complete derivation for the program that sums 1 through n :

$$S \equiv s := 0; i := 0; \text{Loop}; \text{skip} \quad \text{where} \quad \text{Loop} \equiv \text{while } (i < n) \{ i := i + 1; s := s + i \}$$

against the specification

$$\{ n \geq 0 \} \quad S \quad \left\{ s = \frac{n(n+1)}{2} \right\}.$$

The expression language of Section 1 has no division, and the assertion logic need not have one either: since $n(n+1)$ is even, the postcondition is equivalent to

$$Q \equiv 2 \cdot s = n \cdot (n + 1)$$

and we use this form throughout, so that all the reasoning below is in the language of Section 1 extended only by the ordinary laws of integer arithmetic.

The invariant. The one creative step is the invariant for (WHILE). Take

$$I \equiv 0 \leq i \wedge i \leq n \wedge 2 \cdot s = i \cdot (i + 1).$$

The last conjunct says that s holds the sum $1 + \dots + i$; the first two bound the counter. Both bounds are needed, and we check below exactly where each is used.

Pushing the postcondition backwards. Apply (ASSIGN) twice to the loop body, from the back. Writing $I_2 = I[(s+i)/s]$ and $I_1 = I_2[(i+1)/i]$,

$$\begin{aligned} I_2 &= 0 \leq i \wedge i \leq n \wedge 2(s+i) = i(i+1) \\ I_1 &= 0 \leq i+1 \wedge i+1 \leq n \wedge 2(s+i+1) = (i+1)(i+2) \end{aligned}$$

so that (ASSIGN) gives $\vdash \{I_2\} s := s + i \{I\}$ and $\vdash \{I_1\} i := i + 1 \{I_2\}$, and (SEQ) gives

$$\vdash \{I_1\} i := i + 1; s := s + i \{I\}. \tag{1}$$

Likewise for the initialization, writing $J_1 = I[0/i]$ and $J_0 = J_1[0/s]$,

$$\begin{aligned} J_1 &= 0 \leq 0 \wedge 0 \leq n \wedge 2s = 0 \cdot (0 + 1) \\ J_0 &= 0 \leq 0 \wedge 0 \leq n \wedge 2 \cdot 0 = 0 \cdot (0 + 1) \end{aligned}$$

and (ASSIGN) twice with (SEQ) gives

$$\vdash \{J_0\} s := 0; i := 0 \{I\}. \tag{2}$$

The three verification conditions. Three implications remain, each an instance of (CONSEQ). They are the only places where arithmetic is used, and — as Section 8.2 predicts — they are exactly the three blocks that *Blk* of Section 5.2 emits for this program.

(VC1) Entry: $n \geq 0 \rightarrow J_0$. J_0 simplifies to $0 \leq n$, and $0 \leq 0$ and $0 = 0$ are trivially true. This is the only use of the precondition.

(VC2) Body: $I \wedge i < n \rightarrow I_1$. Assume $0 \leq i, i \leq n, 2s = i(i+1)$ and $i < n$. Then $0 \leq i+1$ follows from $0 \leq i$, and $i+1 \leq n$ follows from $i < n$. For the third conjunct,

$$2(s+i+1) = 2s + 2i + 2 = i(i+1) + 2i + 2 = i^2 + 3i + 2 = (i+1)(i+2),$$

using $2s = i(i+1)$ at the second step. Note that the guard $i < n$ is used only for $i+1 \leq n$, and the invariant's conjunct $0 \leq i$ only for $0 \leq i+1$; the arithmetic identity needs neither.

(VC3) Exit: $I \wedge \neg(i < n) \rightarrow Q$. From $\neg(i < n)$ we get $n \leq i$, and with the invariant's $i \leq n$ we get $i = n$. Substituting into $2s = i(i+1)$ gives $2s = n(n+1)$, which is Q . This is the only use of the conjunct $i \leq n$, and it is what makes that conjunct necessary: without it the loop could exit with $i > n$ and nothing would relate s to n .

Assembling the derivation. From (1) and (VC2), (CONSEQ) gives $\vdash \{I \wedge i < n\} i := i+1; s := s+i \{I\}$, which is the premise of (WHILE):

$$\frac{\frac{(1) \quad \models I \wedge i < n \rightarrow I_1}{\vdash \{I \wedge i < n\} i := i+1; s := s+i \{I\}} \text{ (CONSEQ)}}{\vdash \{I\} \text{ Loop } \{I \wedge \neg(i < n)\}} \text{ (WHILE)}$$

From (2) and (VC1), (CONSEQ) gives $\vdash \{n \geq 0\} s := 0; i := 0 \{I\}$. Chaining the two with (SEQ),

$$\vdash \{n \geq 0\} s := 0; i := 0; \text{ Loop } \{I \wedge \neg(i < n)\},$$

and appending $\vdash \{Q\} \text{ skip } \{Q\}$ by (SKIP) and (SEQ), a final (CONSEQ) with (VC3) yields

$$\vdash \{n \geq 0\} S \{2s = n(n+1)\}.$$

By Theorem 8.1 the triple is valid.

The same proof as a proof outline. Derivations are tedious to read, and in practice one writes the same information as an *annotated program*, in which each assertion is the precondition of the statement that follows it. The outline below is the derivation above, and can be checked line by line: each adjacent pair is either an instance of an axiom or an implication marked $\Rightarrow$.

$$\begin{array}{l} \{n \geq 0\} \\ \Rightarrow \{J_0\} \\ s := 0; \\ \{J_1\} \\ i := 0; \\ \{I\} \\ \text{while } (i < n) \{ \\ \quad \{I \wedge i < n\} \\ \quad \Rightarrow \{I_1\} \\ \quad i := i + 1; \\ \quad \{I_2\} \\ \quad s := s + i \\ \quad \{I\} \\ \quad \}; \\ \{I \wedge \neg(i < n)\} \\ \Rightarrow \{2s = n(n+1)\} \\ \text{skip} \\ \{2s = n(n+1)\} \end{array} \quad \begin{array}{l} (VC1) \\ \\ \\ \\ \\ \\ \\ \\ \\ (VC2) \\ \\ \\ (VC3) \end{array}$$

Where the invariant sits. Section 8.4 said that any usable invariant lies between the strongest and the weakest. For this loop, with $P_0 \equiv (s = 0 \wedge i = 0 \wedge n \geq 0)$ the state on entry,

$$sp(P_0, \cdot) \text{ gives } 0 \leq i \leq n \wedge 2s = i(i+1),$$

which is our I exactly: the invariant chosen here *is* the strongest one. At the other extreme, $wlp(\text{Loop}, Q)$ is the weakest, and contains for instance every state with $i > n$, from which the loop exits at once — states the program never reaches, which is why the weakest invariant is rarely the one a person writes. Any I' with $I \Rightarrow I' \Rightarrow wlp(\text{Loop}, Q)$ would do equally well; for example $0 \leq i \leq n \wedge 2s = i(i+1)$ weakened by dropping $0 \leq i$ still proves the program, since (VC2) needs $0 \leq i$ only to re-derive $0 \leq i+1$, which the weakened invariant no longer asserts and therefore no longer has to prove. Dropping $i \leq n$, by contrast, breaks (VC3) irreparably.

8.6 Remarks

The proof system and VC agree. Theorem 8.2 extends to the language with calls: if every block of $VC(\Pi, P, Q)$ from Section 6.5 is valid, then $\Gamma_\Pi \vdash \{P_m\} S_m \{Q_m\}$ is derivable for each m , with (CALL) used at each call site and (INV) immediately after it. Conversely a derivation yields annotations and contracts for which the blocks are valid. Theorem 6.3 and Theorem 8.3 are therefore two packagings of one argument, and it is no accident that both are proved by induction on the height of a big-step derivation.

Heaps. Rules for the language of Section 7 can be given in the same style, using the array encoding. The two dereferencing statements have backward axioms

$$\vdash \{y \neq \text{nil} \wedge Q[H_f[y]/x]\} x := y.f \{Q\} \quad \vdash \{y \neq \text{nil} \wedge Q[H_f[y \mapsto e]/H_f]\} y.f := e \{Q\}$$

reading H_f as an ordinary variable of array sort, so that (ASSIGN) applies unchanged. What does not survive is (INV): the constancy rule is useless once every heap statement writes the single variable H_f , and recovering a usable frame rule is exactly the problem that separation logic solves, and that Section 7 solves differently, by making the modifies clause explicit and the frame a definition. That comparison is a natural next topic and we leave it here.

9 Abstract Interpretation

Sections 5 through 8 all share a shape: the user supplies an invariant for every loop, and a machine checks it. Abstract interpretation inverts this. Nothing is supplied; instead the invariant is *computed*, by solving a system of equations — the same equations that define the semantics of the program, but interpreted over a lattice of *approximate* descriptions of states rather than over sets of states. What is lost is precision: the computed invariant is only as strong as the chosen lattice can express. What is gained is that nobody has to write it.

This section does two things. First it recasts the semantics of while programs as a least fixpoint of a system of equations, which is the form abstract interpretation needs. Then it replaces the lattice in which those equations are solved, and shows that the solution of the new system over-approximates the old one. We work throughout with programs without method calls.

9.1 Complete lattices and least fixpoints

Everything in this section rests on one theorem about monotone maps on complete lattices. We state the little order theory we need and prove it.

Let $(L, \sqsubseteq)$ be a partially ordered set. For $S \subseteq L$, an element $u \in L$ is an *upper bound* of S if $x \sqsubseteq u$ for all $x \in S$, and a *least upper bound* (or *join*) of S , written $\bigsqcup S$, if in addition $u \sqsubseteq u'$ for every upper bound u' of S . *Lower bounds* and the *greatest lower bound* (or *meet*) $\bigsqcap S$ are defined dually. Least upper bounds are unique when they exist, by antisymmetry.

$(L, \sqsubseteq)$ is a *complete lattice* if $\bigsqcup S$ exists for every $S \subseteq L$. Taking $S = \emptyset$ and $S = L$ gives a least element $\perp = \bigsqcup \emptyset$ and a greatest element $\top = \bigsqcap L$.

Lemma 9.1 *In a complete lattice every $S \subseteq L$ also has a greatest lower bound, namely $\bigsqcap S = \bigsqcup \{y \in L : y \text{ is a lower bound of } S\}$.*

Proof. Let B be the set of lower bounds of S and $a = \bigsqcup B$. Every $x \in S$ is an upper bound of B , so $a \sqsubseteq x$ by leastness of the join; hence $a \in B$, i.e. a is a lower bound of S . And any lower bound y of S lies in B , so $y \sqsubseteq a$. Thus a is the greatest lower bound. ■

The examples we need are: $(\mathcal{P}(\Sigma), \sqsubseteq)$, with $\bigsqcup = \bigcup$ and $\bigsqcap = \bigcap$; any finite lattice, such as the sign lattice of Section 9.5; and the pointwise product $A \rightarrow L$ of a complete lattice L , ordered by $m_1 \sqsubseteq m_2$ iff $m_1(a) \sqsubseteq m_2(a)$ for all a , in which joins and meets are computed pointwise.

A map $f : L \rightarrow L$ is *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. Call x a *pre-fixpoint* of f if $f(x) \sqsubseteq x$, a *post-fixpoint* if $x \sqsubseteq f(x)$, and a *fixpoint* if $f(x) = x$.

Theorem 9.2 (Knaster–Tarski) *Let $(L, \sqsubseteq)$ be a complete lattice and $f : L \rightarrow L$ monotone. Then f has a least fixpoint $\text{lfp} f$ and a greatest fixpoint $\text{gfp} f$, and*

$$\text{lfp} f = \bigsqcap \{x : f(x) \sqsubseteq x\}, \quad \text{gfp} f = \bigsqcup \{x : x \sqsubseteq f(x)\}.$$

Moreover $\text{lfp} f$ is the least pre-fixpoint of f : if $f(x) \sqsubseteq x$ then $\text{lfp} f \sqsubseteq x$.

Proof. Let $P = \{x : f(x) \sqsubseteq x\}$ be the set of pre-fixpoints and $a = \bigsqcap P$, which exists by Lemma 9.1.

First, $a \in P$. Let $x \in P$. Then $a \sqsubseteq x$, so $f(a) \sqsubseteq f(x)$ by monotonicity, and $f(x) \sqsubseteq x$ since $x \in P$; hence $f(a) \sqsubseteq x$. As x was an arbitrary element of P , $f(a)$ is a lower bound of P , and therefore $f(a) \sqsubseteq \bigsqcap P = a$. So $a \in P$.

Second, a is a fixpoint. From $f(a) \sqsubseteq a$ and monotonicity, $f(f(a)) \sqsubseteq f(a)$, so $f(a) \in P$, whence $a = \bigsqcap P \sqsubseteq f(a)$. With the previous paragraph and antisymmetry, $f(a) = a$.

Third, a is least among fixpoints, and indeed least among pre-fixpoints: every fixpoint is a pre-fixpoint, so lies in P , and $a = \bigsqcap P$ is below every element of P . This is the final claim of the theorem as well.

The statement for gfp is the dual argument, carried out in the complete lattice $(L, \supseteq)$. ■

The last sentence of the theorem is the one we use most, and it is worth isolating. It is the principle of *fixpoint induction*, sometimes called Park induction:

$$f(x) \sqsubseteq x \implies \text{lfp } f \sqsubseteq x. \quad (\text{P})$$

To prove that a least fixpoint is below something, exhibit that something as a pre-fixpoint. No iteration is involved, and in particular nothing need be known about how many steps the fixpoint takes to form. Every comparison of two fixpoints below — in Theorem 9.5 and in Theorem 9.6 — is an application of (P).

Knaster–Tarski says nothing about *computing* $\text{lfp } f$. For that we need a stronger hypothesis. Call f *continuous* if it preserves joins of nonempty increasing chains: $f(\bigsqcup_n x_n) = \bigsqcup_n f(x_n)$ whenever $x_0 \sqsubseteq x_1 \sqsubseteq \dots$. Continuous maps are monotone.

Theorem 9.3 (Kleene) *If f is continuous on a complete lattice then $\text{lfp } f = \bigsqcup_{n \geq 0} f^n(\perp)$.*

Proof. The sequence is increasing: $\perp \sqsubseteq f(\perp)$, and $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$ follows by monotonicity and induction. Let $a = \bigsqcup_n f^n(\perp)$. By continuity $f(a) = \bigsqcup_n f^{n+1}(\perp) = a$, the last step because adjoining $f^0(\perp) = \perp$ to a set does not change its join. So a is a fixpoint. If $f(x) \sqsubseteq x$ then $f^n(\perp) \sqsubseteq x$ for every n , by induction ($\perp \sqsubseteq x$, and $f^n(\perp) \sqsubseteq x$ gives $f^{n+1}(\perp) \sqsubseteq f(x) \sqsubseteq x$), so $a \sqsubseteq x$. Hence $a = \text{lfp } f$ by Theorem 9.2. ■

Say that L has *finite height* if every strictly increasing chain in L is finite. Then the Kleene sequence $f^n(\perp)$ is eventually constant, so $\text{lfp } f = f^N(\perp)$ for some finite N , and the fixpoint can be *computed* by iterating f from $\perp$ until nothing changes. This is the engine of Section 9.3.

9.2 The collecting semantics

Write Σ for the set of all stores over the variables X . The set $\mathcal{P}(\Sigma)$ of *sets* of stores, ordered by $\subseteq$, is a complete lattice, with $\cup$ as join, $\cap$ as meet, $\emptyset$ as bottom and Σ as top. The move from a single store to a set of stores is the whole idea: a set of stores is a *property*, and the semantics we are about to define propagates properties rather than values. It is called the *collecting* semantics because $\llbracket S \rrbracket(C)$ collects all the stores the program can reach from a store in C .

For a Boolean expression b write $\llbracket b \rrbracket = \{\sigma \in \Sigma : \text{eval}(\sigma, b) = T\}$, and define $\llbracket S \rrbracket : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ by

$$\begin{aligned} \llbracket \text{skip} \rrbracket(C) &= C \\ \llbracket x := e \rrbracket(C) &= \{ \sigma[x \mapsto \text{eval}(\sigma, e)] : \sigma \in C \} \\ \llbracket \text{assume } b \rrbracket(C) &= C \cap \llbracket b \rrbracket \\ \llbracket S_1; S_2 \rrbracket(C) &= \llbracket S_2 \rrbracket(\llbracket S_1 \rrbracket(C)) \\ \llbracket \text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\} \rrbracket(C) &= \llbracket S_1 \rrbracket(C \cap \llbracket b \rrbracket) \cup \llbracket S_2 \rrbracket(C \cap \llbracket \neg b \rrbracket) \\ \llbracket \text{while } (b) \{S\} \rrbracket(C) &= L_C \cap \llbracket \neg b \rrbracket \end{aligned}$$

where, in the last clause, L_C is the *least solution* of the equation

$$X = C \cup \llbracket S \rrbracket(X \cap \llbracket b \rrbracket). \quad (*)$$

Everything except the loop is a direct definition by structural recursion. The loop is the one place where an equation appears, and it is worth saying what X stands for: it is the set of stores that occur *at the head of the loop*, on any iteration. Equation (*) says exactly that such a store either came in from outside the loop (the C disjunct) or is the result of running the body once on a loop-head store that passed the test (the other disjunct). A program with k loops thus gives rise to k equations, one per loop head; for a single loop we have written the system as a single equation.

Why a least solution exists, and why “least”. Equation $(*)$ has many solutions in general — Σ itself usually solves it — and we must say which one is the semantics. Define $F_C : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ by $F_C(X) = C \cup \llbracket S \rrbracket(X \cap \llbracket b \rrbracket)$, so that $(*)$ reads $X = F_C(X)$.

Lemma 9.4 *For every S , the map $\llbracket S \rrbracket$ is monotone and preserves arbitrary unions, that is,*

$$\llbracket S \rrbracket\left(\bigcup_i C_i\right) = \bigcup_i \llbracket S \rrbracket(C_i).$$

Consequently every F_C is monotone, and preserves unions of nonempty chains.

Proof. By induction on S . For the atomic statements both properties are immediate from the definitions, since each is a direct image or an intersection with a fixed set; composition, union and intersection with a fixed set preserve both properties; and for the loop, $L_{\bigcup_i C_i} = \bigcup_i L_{C_i}$ because, as the Kleene description below shows, L_C is built from C by operations that preserve unions. ■

Monotonicity is all that is needed for existence: by Theorem 9.2, F_C has a least fixpoint, and it is the least pre-fixpoint,

$$\text{lfp} F = \bigcap \{ Z : F(Z) \subseteq Z \}. \quad (\dagger)$$

We set $L_C = \text{lfp} F_C$. Preservation of chain unions gives in addition the Kleene description of Theorem 9.3,

$$L_C = \bigcup_{n \geq 0} F_C^n(\emptyset), \quad (\ddagger)$$

which is the iterative reading: $F_C^n(\emptyset)$ is the set of stores reachable at the loop head within n iterations, and L_C their union over all n . The *least* solution is the right one precisely because of this: larger solutions contain stores that no execution produces. The largest solution of $(*)$ is of no use — it is typically Σ — and this is the same distinction, in another guise, as the one between μ and ν in Section 8.4.

Theorem 9.5 *For every statement S and every $C \subseteq \Sigma$,*

$$\llbracket S \rrbracket(C) = \{ \sigma' : \langle S, \sigma \rangle \Downarrow \sigma' \text{ for some } \sigma \in C \} = \text{sp}(C, S).$$

Proof. Induction on S ; the atomic, sequencing and conditional cases are immediate from the corresponding rules of Section 6.2. For the loop, write R for the right-hand side and $L = L_C$.

($\supseteq$) Suppose $\langle \text{while } (b) \{ S \}, \sigma \rangle \Downarrow \sigma'$ with $\sigma \in C$. By induction on the height of that derivation one shows that the store at the head of each iteration lies in L : the first is $\sigma \in C \subseteq F_C(L) = L$, and if a loop-head store $\tau \in L$ passes the test then the next loop-head store lies in $\llbracket S \rrbracket(L \cap \llbracket b \rrbracket) \subseteq L$ by the induction hypothesis for S and $(*)$. The computation ends when the test fails, so $\sigma' \in L \cap \llbracket \neg b \rrbracket$.

($\subseteq$) By ($\ddagger$) it suffices to show, by induction on n , that every $\sigma'' \in F_C^n(\emptyset)$ is a loop-head store of some computation started in C ; then a $\sigma'' \in L$ failing b terminates that computation by rule (WHILE-F), giving $\sigma'' \in R$. For $n = 0$ this is vacuous, and the step is again the induction hypothesis for S together with rule (WHILE-T). ■

So the collecting semantics is the strongest postcondition of Section 8.4, and L_C is the *strongest invariant* of the loop: the smallest set of stores closed under the body and containing the entry states. Every invariant in the sense of Section 5 is a superset of it. Abstract interpretation computes a superset too — just one that is computable.

9.3 Abstract domains and the abstract equations

Fix a complete lattice $(D, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$, the *abstract domain*, together with a monotone map

$$\gamma : D \longrightarrow \mathcal{P}(\Sigma)$$

called the *concretization*: $\gamma(d)$ is the set of stores that the abstract element d describes. Monotonicity says that a larger abstract element describes more stores. Nothing else is assumed — in particular we do *not* assume that every set of stores has a best abstract description, which is an additional hypothesis taken up later.

An *abstract interpretation* consists of D , γ , and for each atomic statement a monotone *abstract transfer function*:

$$\text{asgn}_{x,e}^\# : D \rightarrow D, \quad \text{filt}_b^\# : D \rightarrow D$$

required to be *locally sound*, that is, for every $d \in D$:

$$\llbracket x := e \rrbracket^\#(\gamma(d)) \subseteq \gamma(\text{asgn}_{x,e}^\#(d)), \quad \gamma(d) \cap \llbracket b \rrbracket \subseteq \gamma(\text{filt}_b^\#(d)). \quad (\S)$$

Local soundness is a condition on *one* statement at a time, and it is deliberately an inequality: the abstract function may lose information, but must never lose a store. Note that $\text{filt}_b^\# = \text{id}$ always satisfies the second condition; a filter that does nothing is sound, merely useless.

Now repeat the definitions of Section 9.2 verbatim, replacing $\cup$ by $\sqcup$, intersection with $\llbracket b \rrbracket$ by $\text{filt}_b^\#$, and the atomic clauses by their abstract counterparts:

$$\begin{aligned} \llbracket \text{skip} \rrbracket^\#(d) &= d \\ \llbracket x := e \rrbracket^\#(d) &= \text{asgn}_{x,e}^\#(d) \\ \llbracket \text{assume } b \rrbracket^\#(d) &= \text{filt}_b^\#(d) \\ \llbracket S_1; S_2 \rrbracket^\#(d) &= \llbracket S_2 \rrbracket^\#(\llbracket S_1 \rrbracket^\#(d)) \\ \llbracket \text{if } b \text{ then } \{S_1\} \text{ else } \{S_2\} \rrbracket^\#(d) &= \llbracket S_1 \rrbracket^\#(\text{filt}_b^\#(d)) \sqcup \llbracket S_2 \rrbracket^\#(\text{filt}_{\neg b}^\#(d)) \\ \llbracket \text{while } (b)\{S\} \rrbracket^\#(d) &= \text{filt}_{\neg b}^\#(L_d^\#) \end{aligned}$$

where $L_d^\#$ is the least solution in D of the *abstract equation*

$$Y = d \sqcup \llbracket S \rrbracket^\#(\text{filt}_b^\#(Y)). \quad (*^\#)$$

Each $\llbracket S \rrbracket^\#$ is monotone, by induction on S , so Theorem 9.2 again supplies the least solution. The equation is the same equation; only the lattice has changed.

Computing the solution. If D has finite height in the sense of Section 9.1, then the Kleene iteration

$$\perp \sqsubseteq G_d(\perp) \sqsubseteq G_d^2(\perp) \sqsubseteq \cdots, \quad G_d(Y) = d \sqcup \llbracket S \rrbracket^\#(\text{filt}_b^\#(Y)),$$

is an increasing chain and therefore stabilizes after finitely many steps, at $L_d^\#$. This is the point of the whole exercise: in $\mathcal{P}(\Sigma)$ the iteration $(\ddagger)$ generally runs forever, while in a finite-height D it terminates, and mechanically. (For domains of infinite height, such as intervals, one accelerates the iteration with a *widening* operator; we do not treat that here.)

9.4 Soundness

Theorem 9.6 (Soundness of the abstract semantics) *Suppose that the abstract transfer functions are monotone and locally sound. Then for every statement S and every $d \in D$ we have*

$$\llbracket S \rrbracket(\gamma(d)) \subseteq \gamma(\llbracket S \rrbracket^\#(d)).$$

Proof. Induction on S . For **skip** there is nothing to prove, and the two atomic cases are exactly the hypotheses $(\S)$.

For $S_1; S_2$, the induction hypothesis for S_1 gives $\llbracket S_1 \rrbracket(\gamma(d)) \subseteq \gamma(\llbracket S_1 \rrbracket^\#(d))$; applying the monotone map $\llbracket S_2 \rrbracket$ (Lemma 9.4) and then the induction hypothesis for S_2 ,

$$\llbracket S_2 \rrbracket(\llbracket S_1 \rrbracket(\gamma(d))) \subseteq \llbracket S_2 \rrbracket(\gamma(\llbracket S_1 \rrbracket^\#(d))) \subseteq \gamma(\llbracket S_2 \rrbracket^\#(\llbracket S_1 \rrbracket^\#(d))).$$

For the conditional, note first that monotonicity of γ gives $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2)$, since each $\gamma(d_i) \subseteq \gamma(d_1 \sqcup d_2)$. Then

$$\llbracket S_1 \rrbracket(\gamma(d) \cap \llbracket b \rrbracket) \subseteq \llbracket S_1 \rrbracket(\gamma(\text{filt}_b^\#(d))) \subseteq \gamma(\llbracket S_1 \rrbracket^\#(\text{filt}_b^\#(d)))$$

using (§), monotonicity of $\llbracket S_1 \rrbracket$ and the induction hypothesis; symmetrically for S_2 ; and the union of the two is contained in γ of the join.

For the loop, the key step is *fixpoint induction*. Write $F(X) = \gamma(d') \cup \dots$ — more precisely let $C = \gamma(d)$, $F(X) = C \cup \llbracket S \rrbracket(X \cap \llbracket b \rrbracket)$ and $G(Y) = d \sqcup \llbracket S \rrbracket^\#(\text{filt}_b^\#(Y))$, with $L = \text{lfp} F$ and $L^\# = \text{lfp} G$. By fixpoint induction (P), to prove $L \subseteq \gamma(L^\#)$ it suffices to show that $\gamma(L^\#)$ is a pre-fixpoint of F , i.e. $F(\gamma(L^\#)) \subseteq \gamma(L^\#)$. Compute:

$$\begin{aligned} F(\gamma(L^\#)) &= C \cup \llbracket S \rrbracket(\gamma(L^\#) \cap \llbracket b \rrbracket) \\ &\subseteq \gamma(d) \cup \llbracket S \rrbracket(\gamma(\text{filt}_b^\#(L^\#))) && \text{by (§) and monotonicity of } \llbracket S \rrbracket \\ &\subseteq \gamma(d) \cup \gamma(\llbracket S \rrbracket^\#(\text{filt}_b^\#(L^\#))) && \text{by the induction hypothesis for } S \\ &\subseteq \gamma(d \sqcup \llbracket S \rrbracket^\#(\text{filt}_b^\#(L^\#))) && \text{by monotonicity of } \gamma \\ &= \gamma(G(L^\#)) = \gamma(L^\#) && \text{since } L^\# \text{ is a fixpoint of } G. \end{aligned}$$

Hence $L \subseteq \gamma(L^\#)$, and finally

$$\llbracket \text{while } (b)\{S\} \rrbracket(C) = L \cap \llbracket \neg b \rrbracket \subseteq \gamma(L^\#) \cap \llbracket \neg b \rrbracket \subseteq \gamma(\text{filt}_{\neg b}^\#(L^\#)) = \gamma(\llbracket \text{while } (b)\{S\} \rrbracket^\#(d))$$

using (§) once more. ■

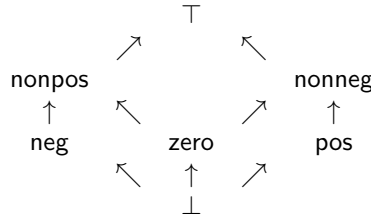
Two things deserve emphasis. First, the hypotheses are entirely *local*: one checks soundness of each transfer function for one statement, and the theorem lifts this to whole programs, loops included. Second, the loop case never computes anything — it does not need to know how many iterations either fixpoint takes — because (P) characterizes *lfp* as the least pre-fixpoint, and $\gamma(L^\#)$ is shown to be one. This is the standard way to compare two fixpoints and is worth remembering.

Corollary 9.6.1 *If $C \subseteq \gamma(d)$ then every store reachable at the head of the loop $\text{while } (b)\{S\}$ from a store in C lies in $\gamma(L_d^\#)$. That is, $\gamma(L_d^\#)$ is a loop invariant in the sense of Section 5 — computed, not supplied.*

9.5 The sign domain

We instantiate all of this with the simplest non-trivial domain: one that remembers, for each variable, only its sign.

Signs. Let **Sign** be the following seven-element lattice.



Thus $\text{neg} \sqcup \text{zero} = \text{nonpos}$, $\text{zero} \sqcup \text{pos} = \text{nonneg}$, $\text{neg} \sqcup \text{pos} = \top$ and $\text{nonpos} \sqcap \text{nonneg} = \text{zero}$. Its concretization $\gamma_1 : \text{Sign} \rightarrow \mathcal{P}(\mathbb{Z})$ is

$$\begin{aligned} \gamma_1(\perp) &= \emptyset, & \gamma_1(\text{neg}) &= \{z < 0\}, & \gamma_1(\text{zero}) &= \{0\}, & \gamma_1(\text{pos}) &= \{z > 0\}, \\ \gamma_1(\text{nonpos}) &= \{z \leq 0\}, & \gamma_1(\text{nonneg}) &= \{z \geq 0\}, & \gamma_1(\top) &= \mathbb{Z}. \end{aligned}$$

The lattice has height 3, so it satisfies the finite-height condition of Section 9.3.

The abstract domain. Take $D = X \rightarrow \text{Sign}$ with the pointwise order, which is again a complete lattice of finite height $3|X|$, and

$$\gamma(m) = \{ \sigma \in \Sigma : \sigma(x) \in \gamma_1(m(x)) \text{ for every } x \in X \}.$$

Note $\gamma(m) = \emptyset$ as soon as $m(x) = \perp$ for some x ; several abstract elements therefore describe the empty set, which is harmless. γ is monotone because γ_1 is.

Abstract arithmetic. Define $aeval^\sharp(m, e) \in \text{Sign}$ by recursion on e : $aeval^\sharp(m, x) = m(x)$, and $aeval^\sharp(m, c)$ is **neg**, **zero** or **pos** according to the sign of the constant c . On the three basic signs, multiplication follows the rule of signs and is exact; addition is where information gets lost. The two tables are

+	neg	zero	pos	×	neg	zero	pos
neg	neg	neg	⊤	neg	pos	zero	neg
zero	neg	zero	pos	zero	zero	zero	zero
pos	⊤	pos	pos	pos	neg	zero	pos

and extending to the compound signs by $s \hat{+} (s_1 \sqcup s_2) = (s \hat{+} s_1) \sqcup (s \hat{+} s_2)$, and similarly for $\times$ and for $-$ (which is $+$ after negating the sign of its second argument). The single entry **neg** $\hat{+}$ **pos** = $\top$ is the source of essentially all imprecision in the analyses below: the domain simply cannot represent the sign of a sum of a negative and a positive quantity.

Transfer functions.

$$asgn_{x,e}^\sharp(m) = m[x \mapsto aeval^\sharp(m, e)]$$

and for the filter we give a few exact cases and fall back on the identity otherwise:

$$\begin{aligned} filt_{x < 0}^\sharp(m) &= m[x \mapsto m(x) \sqcap \text{neg}] \\ filt_{x \leq 0}^\sharp(m) &= m[x \mapsto m(x) \sqcap \text{nonpos}] \\ filt_{x = 0}^\sharp(m) &= m[x \mapsto m(x) \sqcap \text{zero}] \\ filt_b^\sharp(m) &= m \quad \text{for every other } b \end{aligned}$$

(together with the symmetric cases for $x > 0$, $x \geq 0$, and conjunctions handled by composing filters). Local soundness of the assignment is the statement that $aeval^\sharp$ over-approximates $eval$, which follows from the tables by induction on e ; local soundness of the filter is immediate, and in the default case it is the observation that $C \cap \llbracket b \rrbracket \subseteq C$.

9.6 A worked example

Take the program of Section 8.5 again, and give it the precondition $n \geq 0$:

$s := 0; i := 0; \text{ while } (i < n) \{ i := i + 1; s := s + i \}; \text{ skip}$

The entry abstract state is $d_0 = \{ n \mapsto \text{nonneg}, i \mapsto \top, s \mapsto \top \}$, whose concretization is exactly $\{ \sigma : \sigma(n) \geq 0 \}$. The two initializations give

$$d = \llbracket s := 0; i := 0 \rrbracket^\sharp(d_0) = \{ n \mapsto \text{nonneg}, i \mapsto \text{zero}, s \mapsto \text{zero} \}.$$

Now solve the abstract equation $(*)^\sharp$ at the loop head by Kleene iteration. The guard is $i < n$, a comparison between two variables, so our filter is the identity on it; the body is $i := i + 1; s := s + i$. Writing

each abstract state as a triple (n, i, s) :

$$\begin{aligned}
Y_0 &= \perp \\
Y_1 = G(Y_0) &= d = (\text{nonneg}, \text{zero}, \text{zero}) \\
Y_2 = G(Y_1) &= d \sqcup \llbracket \text{body} \rrbracket^\sharp(\text{nonneg}, \text{zero}, \text{zero}) = (\text{nonneg}, \text{zero}, \text{zero}) \sqcup (\text{nonneg}, \text{pos}, \text{pos}) \\
&= (\text{nonneg}, \text{nonneg}, \text{nonneg}) \\
Y_3 = G(Y_2) &= d \sqcup \llbracket \text{body} \rrbracket^\sharp(\text{nonneg}, \text{nonneg}, \text{nonneg}) = (\text{nonneg}, \text{zero}, \text{zero}) \sqcup (\text{nonneg}, \text{pos}, \text{pos}) \\
&= (\text{nonneg}, \text{nonneg}, \text{nonneg}) = Y_2
\end{aligned}$$

The two body computations are: from $(\text{nonneg}, \text{zero}, \text{zero})$, the assignment $i := i + 1$ gives $i \mapsto \text{zero} \hat{+} \text{pos} = \text{pos}$, and then $s := s + i$ gives $s \mapsto \text{zero} \hat{+} \text{pos} = \text{pos}$; from $(\text{nonneg}, \text{nonneg}, \text{nonneg})$, $i := i + 1$ gives $i \mapsto \text{nonneg} \hat{+} \text{pos} = \text{pos}$ and then $s := s + i$ gives $s \mapsto \text{nonneg} \hat{+} \text{pos} = \text{pos}$. The iteration has stabilized after three steps, so

$$L_d^\sharp = (\text{nonneg}, \text{nonneg}, \text{nonneg}), \quad \text{that is,} \quad \gamma(L_d^\sharp) = \{ \sigma : \sigma(n) \geq 0 \wedge \sigma(i) \geq 0 \wedge \sigma(s) \geq 0 \}.$$

Since the exit filter for $\neg(i < n)$ is the identity, the analysis concludes

$$\{ n \geq 0 \} \ S \ \{ s \geq 0 \wedge i \geq 0 \}$$

and, by Corollary 9.6.1, $i \geq 0 \wedge s \geq 0 \wedge n \geq 0$ is a loop invariant. Nobody wrote it down.

What the example shows. Compare this with Section 8.5, where the invariant $0 \leq i \leq n \wedge 2s = i(i+1)$ was supplied by hand and the proof established the exact postcondition $2s = n(n+1)$. The sign analysis is fully automatic and establishes $s \geq 0$ — and it cannot possibly do better, because neither $i \leq n$ nor $2s = i(i+1)$ is expressible in D at all. The conjunct $i \leq n$ relates two variables, and D is *non-relational*: it records a property of each variable separately, so no element of D has γ equal to $\{ \sigma : \sigma(i) \leq \sigma(n) \}$. The invariant $2s = i(i+1)$ is beyond any linear domain.

This is the characteristic trade of abstract interpretation. A domain is chosen; it fixes, once and for all, the vocabulary in which invariants can be stated; and within that vocabulary the strongest invariant — or at least a sound one — is computed without human intervention. Richer domains (intervals, octagons, convex polyhedra) widen the vocabulary at increasing cost, and the fixpoint iteration for domains of infinite height needs the widening operators mentioned in Section 9.3.

A remark on precision. The computed $L_d^\sharp$ need not be the best abstract element describing the concrete $L_{\gamma(d)}$: it is merely *some* sound one. Here it happens to be optimal, but in general the losses at each individual transfer function compound. Making precise what “the best abstract element describing a set of stores” means, and when the abstract fixpoint is as precise as that, requires a map in the other direction, from $\mathcal{P}(\Sigma)$ to D , and a condition relating the two. That is the subject of Galois connections, which we take up separately.

9.7 Galois connections

Section 9.3 asked only for a concretization γ , and Section 9.6 ended with a question that γ alone cannot answer: what is the *best* abstract element describing a given set of stores? For that we need a map in the other direction.

Theorem 9.7 *Let $(\mathcal{P}(\Sigma), \subseteq)$ and $(D, \sqsubseteq)$ be complete lattices and let $\alpha : \mathcal{P}(\Sigma) \rightarrow D$ and $\gamma : D \rightarrow \mathcal{P}(\Sigma)$ be any maps. The following are equivalent.*

- (i) *For all $C \subseteq \Sigma$ and $d \in D$: $\alpha(C) \sqsubseteq d \iff C \subseteq \gamma(d)$.*
- (ii) *α and γ are monotone, $C \subseteq \gamma(\alpha(C))$ for every C , and $\alpha(\gamma(d)) \sqsubseteq d$ for every d .*

When these hold we call (α, γ) a Galois connection, written $(\mathcal{P}(\Sigma), \subseteq) \overset{\alpha}{\underset{\gamma}{\rightleftarrows}} (D, \sqsubseteq)$, and call α the abstraction map.

Proof. (i) $\Rightarrow$ (ii). Taking $d = \alpha(C)$ in (i) and using $\alpha(C) \sqsubseteq \alpha(C)$ gives $C \subseteq \gamma(\alpha(C))$; dually, taking $C = \gamma(d)$ gives $\alpha(\gamma(d)) \sqsubseteq d$. For monotonicity of α , let $C_1 \subseteq C_2$; then $C_1 \subseteq C_2 \subseteq \gamma(\alpha(C_2))$, so by (i) $\alpha(C_1) \sqsubseteq \alpha(C_2)$. Monotonicity of γ is dual.

(ii) $\Rightarrow$ (i). If $\alpha(C) \sqsubseteq d$ then $\gamma(\alpha(C)) \subseteq \gamma(d)$ by monotonicity, and $C \subseteq \gamma(\alpha(C))$, so $C \subseteq \gamma(d)$. Conversely if $C \subseteq \gamma(d)$ then $\alpha(C) \sqsubseteq \alpha(\gamma(d)) \sqsubseteq d$. ■

Condition (i) is the compact form; condition (ii) is the one to keep in mind. Read $C \subseteq \gamma(\alpha(C))$ as “abstracting loses nothing, it only adds” and $\alpha(\gamma(d)) \sqsubseteq d$ as “concretizing and re-abstracting does not lose either”.

Corollary 9.7.1 *In a Galois connection, $\alpha(C)$ is the least abstract element describing C :*

$$\alpha(C) = \bigsqcap \{d \in D : C \subseteq \gamma(d)\},$$

and dually $\gamma(d) = \bigcup \{C : \alpha(C) \sqsubseteq d\}$. Hence each of α, γ determines the other.

Proof. By (i), the set on the right is $\{d : \alpha(C) \sqsubseteq d\}$, whose greatest lower bound is $\alpha(C)$. The dual statement is symmetric. ■

Corollary 9.7.2 *α preserves all joins and γ all meets: $\alpha(\bigcup_i C_i) = \bigsqcup_i \alpha(C_i)$ and $\gamma(\bigsqcap_i d_i) = \bigcap_i \gamma(d_i)$. In particular $\alpha(\emptyset) = \perp$.*

Proof. For any d : $\alpha(\bigcup_i C_i) \sqsubseteq d$ iff $\bigcup_i C_i \subseteq \gamma(d)$ iff $C_i \subseteq \gamma(d)$ for all i iff $\alpha(C_i) \sqsubseteq d$ for all i iff $\bigsqcup_i \alpha(C_i) \sqsubseteq d$. Two elements below exactly the same elements are equal. The claim for γ is dual, and $\alpha(\emptyset) = \perp$ is the case of the empty union. ■

If moreover $\alpha(\gamma(d)) = d$ for every d — equivalently, γ is injective — the connection is called a *Galois insertion*. Failure of this is not a defect but redundancy: several abstract elements describe the same set, as happens in Section 9.5, where every map m with some $m(x) = \perp$ has $\gamma(m) = \emptyset$. One obtains an insertion by identifying all such elements.

The sign domain is a Galois connection. Define $\alpha_1 : \mathcal{P}(\mathbb{Z}) \rightarrow \text{Sign}$ by $\alpha_1(Z) = \bigsqcup \{ \text{sgn}(z) : z \in Z \}$, where $\text{sgn}(z)$ is neg, zero or pos, and lift it to stores by

$$\alpha(C)(x) = \alpha_1(\{ \sigma(x) : \sigma \in C \}).$$

That (α, γ) satisfies (i) is a direct check: $\alpha(C) \sqsubseteq m$ holds iff for every x every value $\sigma(x)$ taken in C has $\text{sgn}(\sigma(x)) \sqsubseteq m(x)$, which is exactly $C \subseteq \gamma(m)$. It is not an insertion, for the reason just given.

Best transfer functions. With α in hand we can say what the most precise sound abstract transfer function is. For a concrete monotone $f : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$, put

$$f^{\text{best}} = \alpha \circ f \circ \gamma.$$

Theorem 9.8 *Let $f^\sharp : D \rightarrow D$ be monotone. Then $f^\sharp$ is locally sound in the sense of (§), i.e. $f(\gamma(d)) \subseteq \gamma(f^\sharp(d))$ for all d , if and only if $f^{\text{best}} \sqsubseteq f^\sharp$ pointwise. In particular f^{best} is itself locally sound, and is the least such function.*

Proof. By Theorem 9.7(i) applied with $C = f(\gamma(d))$:

$$f(\gamma(d)) \subseteq \gamma(f^\sharp(d)) \iff \alpha(f(\gamma(d))) \sqsubseteq f^\sharp(d) \iff f^{\text{best}}(d) \sqsubseteq f^\sharp(d).$$

■

So the design of an abstract interpretation splits cleanly in two. The domain and its Galois connection fix f^{best} once and for all; an implementation may then use any monotone $f^\sharp$ above it, trading precision for ease of computation. The abstract addition of Section 9.5 is in fact best, but the abstract evaluation of a compound expression is not: with $m(y) = \top$,

$$aeval^\sharp(m, y - y) = \top \hat{=} \top = \top, \quad \text{whereas} \quad \alpha_1(\{z - z : z \in \mathbb{Z}\}) = \text{zero}.$$

Evaluating an expression subterm by subterm forgets that the two occurrences of y denote the same value. This is the first and most common source of imprecision, and it has nothing to do with loops.

Fixpoints need not transfer exactly. The second source does concern loops, and it survives even when every transfer function is best.

Theorem 9.9 (Fixpoint transfer) *Let F be monotone on $\mathcal{P}(\Sigma)$ and $F^\sharp$ monotone on D .*

- (a) *If $\alpha \circ F \sqsubseteq F^\sharp \circ \alpha$ pointwise, then $\alpha(\text{lfp} F) \sqsubseteq \text{lfp} F^\sharp$.*
- (b) *If $\alpha \circ F = F^\sharp \circ \alpha$ pointwise, F is continuous and D has finite height, then $\alpha(\text{lfp} F) = \text{lfp} F^\sharp$.*

Proof. (a) Put $x = \gamma(\text{lfp} F^\sharp)$. Then

$$\alpha(F(x)) \sqsubseteq F^\sharp(\alpha(\gamma(\text{lfp} F^\sharp))) \sqsubseteq F^\sharp(\text{lfp} F^\sharp) = \text{lfp} F^\sharp,$$

using the hypothesis, then $\alpha \circ \gamma \sqsubseteq \text{id}$ with monotonicity of $F^\sharp$. By Theorem 9.7(i) this says $F(x) \sqsubseteq \gamma(\text{lfp} F^\sharp) = x$, so x is a pre-fixpoint of F , and fixpoint induction (P) gives $\text{lfp} F \sqsubseteq x$, that is, $\alpha(\text{lfp} F) \sqsubseteq \text{lfp} F^\sharp$.

(b) We show $\alpha(F^n(\emptyset)) = (F^\sharp)^n(\perp)$ for every n , by induction. For $n = 0$ this is $\alpha(\emptyset) = \perp$, which is Corollary 9.7.2. For the step, $\alpha(F^{n+1}(\emptyset)) = F^\sharp(\alpha(F^n(\emptyset))) = F^\sharp((F^\sharp)^n(\perp))$ by the hypothesis and the induction hypothesis. Now $\text{lfp} F = \bigcup_n F^n(\emptyset)$ by Theorem 9.3, so by Corollary 9.7.2 $\alpha(\text{lfp} F) = \bigsqcup_n \alpha(F^n(\emptyset)) = \bigsqcup_n (F^\sharp)^n(\perp)$, and the right-hand side is $\text{lfp} F^\sharp$ because finite height makes the chain stabilize at the least fixpoint. ■

The inequality in (a) is generally strict, and — this is the point — it can be strict even when every transfer function is the best one. Consider, over the single variable x ,

$$x := 0; \text{ while } (\dots) \{ x := 1 - x \}$$

with a guard the filter cannot exploit, so that $F(X) = \{0\} \cup \llbracket x := 1 - x \rrbracket(X)$ and $F^\sharp(Y) = \text{zero} \sqcup f^{\text{best}}(Y)$. Concretely the loop head collects $\text{lfp} F = \{0, 1\}$, whose abstraction is **nonneg**. Abstractly, the Kleene iteration runs

$$\perp \sqsubseteq \text{zero} \sqsubseteq \text{zero} \sqcup \text{pos} = \text{nonneg} \sqsubseteq \text{zero} \sqcup \top = \top,$$

the last step because $f^{\text{best}}(\text{nonneg}) = \alpha_1(\{1 - z : z \geq 0\}) = \alpha_1(\{z \leq 1\}) = \top$. So $\text{lfp} F^\sharp = \top$ while $\alpha(\text{lfp} F) = \text{nonneg}$.

The two chains of part (b) are therefore

n	0	1	2	3
$\alpha(F^n(\emptyset))$	$\perp$	zero	nonneg	nonneg
$(F^\sharp)^n(\perp)$	$\perp$	zero	nonneg	$\top$

and they part company at $n = 3$. The commutation hypothesis of (b) first fails at $C = \{0, 1\}$: there $\alpha(F(C)) = \alpha(\{0, 1\}) = \text{nonneg}$, whereas $F^\sharp(\alpha(C)) = F^\sharp(\text{nonneg}) = \top$. Abstracting *first* throws away the knowledge that x is 0 or 1 and nothing else, which the concrete step still has; the loss happens at the join, which has merged the even and the odd iterates before the body is applied. No choice of transfer function repairs this — only a richer domain, one able to express “ $x \in \{0, 1\}$ ”.

Where this leaves Section 9.4. Theorem 9.6 was proved with γ alone and remains the operative soundness statement; what the Galois connection adds is a yardstick. It tells us, for a given domain, exactly how precise an analysis could in principle be (f^{best} , and $\alpha(\text{lfp} F)$), and lets us measure a particular analysis against that. It does not make the analysis more precise by itself.

9.8 Constant Propagation

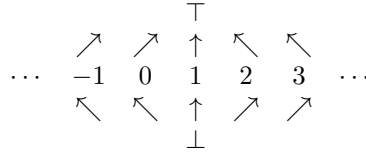
The sign domain of Section 9.5 records, for each variable, which of seven sign properties it has. We now instantiate the framework of this section with a domain that records something different and in a sense orthogonal: whether a variable is known to hold one particular integer. The analysis is *constant propagation*, and it is the oldest dataflow analysis there is. It is also the standard example of a phenomenon we have so far only asserted: that the join in the abstract lattice loses information which no choice of transfer function can recover.

9.8.1 The domain

Let

$$\text{Const} = \{\perp\} \cup \mathbb{Z} \cup \{\top\}$$

ordered by $\perp \sqsubseteq c \sqsubseteq \top$ for every $c \in \mathbb{Z}$, with distinct integers incomparable. This is the *flat lattice* over $\mathbb{Z}$:



It is a complete lattice: the join of a set $S \subseteq \text{Const}$ is $\perp$ if $S \subseteq \{\perp\}$, is c if the non- $\perp$ elements of S are exactly $\{c\}$ for a single integer c , and is $\top$ otherwise. Concretization and abstraction are

$$\gamma_1(\perp) = \emptyset, \quad \gamma_1(c) = \{c\}, \quad \gamma_1(\top) = \mathbb{Z}, \quad \alpha_1(Z) = \begin{cases} \perp & \text{if } Z = \emptyset \\ c & \text{if } Z = \{c\} \\ \top & \text{otherwise,} \end{cases}$$

and (α_1, γ_1) is a Galois connection — indeed a Galois insertion, since $\alpha_1(\gamma_1(d)) = d$ for all three shapes of d . As in Section 9.5, take $D = X \rightarrow \text{Const}$ with the pointwise order, and lift γ_1, α_1 pointwise.

Two features of Const are worth noticing, because they differ from the sign lattice.

- **It has height 2 but is infinite.** Every strictly increasing chain has the form $\perp \sqsubset c \sqsubset \top$ and so has at most three elements. The finite-height condition of Section 9.1 therefore holds, and the Kleene iteration of Section 9.3 terminates — in at most $2|X| + 1$ steps — even though the lattice itself is infinite. Finite *height* is what matters for termination, not finiteness.
- **It is not distributive, in the sense that matters below.** The lattice Const itself is distributive, but the abstract transfer functions built on it are not join-morphisms, and that is what causes trouble in Section 9.8.4.

9.8.2 Transfer functions

Abstract evaluation is exact when it can be:

$$\text{aeval}^\sharp(m, x) = m(x), \quad \text{aeval}^\sharp(m, c) = c,$$

and for a binary operation $\oplus \in \{+, -, \times\}$,

$$d_1 \hat{\oplus} d_2 = \begin{cases} \perp & \text{if } d_1 = \perp \text{ or } d_2 = \perp \\ c_1 \oplus c_2 & \text{if } d_1 = c_1 \in \mathbb{Z} \text{ and } d_2 = c_2 \in \mathbb{Z} \\ \top & \text{otherwise.} \end{cases}$$

The $\perp$ case says that an unreachable state yields an unreachable state, and it matters: without it, an abstract state with some variable $\perp$ would propagate $\top$ instead of $\perp$ and the analysis would lose track of unreachability.

The assignment and filter transfer functions are

$$\text{asgn}_{x,e}^\#(m) = m[x \mapsto \text{aeval}^\#(m, e)], \quad \text{filt}_{x=e}^\#(m) = m[x \mapsto m(x) \sqcap \text{aeval}^\#(m, e)],$$

with $\text{filt}_b^\# = \text{id}$ for every other b . Note that $x = e$ is the only test this domain can exploit: from $x \neq 3$ one learns nothing expressible in **Const**, and from $x < 3$ likewise. (If the filter produces a map with some component $\perp$, the branch is unreachable, which the $\perp$ clause above then propagates.)

These are not the best transfer functions. Local soundness is easy to check, but Theorem 9.8 invites a sharper question: is $\hat{\oplus}$ equal to $\alpha_1 \circ \oplus \circ \gamma_1$? For $+$ it is: $\alpha_1(\{z_1 + z_2 : z_1 \in \gamma_1(d_1), z_2 \in \gamma_1(d_2)\})$ is a singleton only when both arguments are singletons. For $\times$ it is *not*:

$$0 \hat{\times} \top = \top, \quad \text{whereas} \quad \alpha_1(\{0 \cdot z : z \in \mathbb{Z}\}) = \alpha_1(\{0\}) = 0.$$

Multiplying by a known zero yields a known zero whatever the other factor is. The implementation may of course add this case — and real compilers do — which is exactly the latitude Theorem 9.8 describes: any monotone $f^\#$ with $f^{\text{best}} \sqsubseteq f^\#$ is sound, and one chooses how close to f^{best} to get.

9.8.3 Example: propagation through a loop

Consider

`$x := 2; y := 3; i := 0; \text{ while } (i < n) \{ z := x \times y; i := i + 1 \}; \text{ skip}$`

starting from d_0 with every variable $\top$. The initial assignments give $d = \{x \mapsto 2, y \mapsto 3, i \mapsto 0, z \mapsto \top, n \mapsto \top\}$, and the loop-head equation $(*)^\#$ is solved by Kleene iteration. Writing abstract states as tuples (x, y, i, z) and suppressing n , which stays $\top$:

$$\begin{aligned} Y_0 &= \perp \\ Y_1 &= d = (2, 3, 0, \top) \\ Y_2 &= d \sqcup \llbracket \text{body} \rrbracket^\#(2, 3, 0, \top) = (2, 3, 0, \top) \sqcup (2, 3, 1, 6) = (2, 3, \top, \top) \\ Y_3 &= d \sqcup \llbracket \text{body} \rrbracket^\#(2, 3, \top, \top) = (2, 3, 0, \top) \sqcup (2, 3, \top, 6) = (2, 3, \top, \top) = Y_2 \end{aligned}$$

The analysis concludes that $x = 2$ and $y = 3$ throughout the loop, which is correct and useful: a compiler may replace $x \times y$ by the constant 6 inside the body. It concludes nothing about i , which is as it must be, since i genuinely takes many values.

Note what happened to z at the loop head: $0 \sqcup \top$ aside, the join of the entry value $\top$ with the body's value 6 is $\top$, so the invariant says nothing about z ; but inside the body, after the assignment, $z = 6$ holds. The abstract state is attached to a program point, and different points carry different information.

9.8.4 Why the join loses: merge-over-paths versus least fixpoint

Constant propagation is the standard example of a gap that Section 9.7 described abstractly. Consider the loop-free program

`$\text{if } (b) \text{ then } \{ x := 1; y := 2 \} \text{ else } \{ x := 2; y := 1 \}; z := x + y$`

with b a test the filter cannot exploit. Along *either* path, z receives 3. So the collecting semantics gives a set of stores all of which satisfy $z = 3$, and α of it has $z \mapsto 3$.

The abstract semantics does something else. By the clause for conditionals,

$$\llbracket \text{if } \dots \rrbracket^\#(d) = (1, 2) \sqcup (2, 1) = (\top, \top)$$

in the components (x, y) , and then $z := x + y$ gives $\top \hat{+} \top = \top$. The analysis loses the constant.

The reason is visible in one line: the transfer function $f(m) = m[z \mapsto m(x) \hat{+} m(y)]$ is *not a join-morphism*. With $d_1 = (x \mapsto 1, y \mapsto 2)$ and $d_2 = (x \mapsto 2, y \mapsto 1)$,

$$f(d_1) \sqcup f(d_2) = (\top, \top, z \mapsto 3) \quad \sqsubset \quad f(d_1 \sqcup d_2) = (\top, \top, z \mapsto \top).$$

And this is not a defect of our $\hat{+}$: the best transfer function has $\top \hat{+} \top = \alpha_1(\{z_1 + z_2\}) = \top$ as well. The information destroyed is the *correlation* between x and y — that they are $(1, 2)$ or $(2, 1)$ but never $(1, 1)$ — and no non-relational domain can hold it.

MOP and MFP. The two quantities being compared have names. The *merge over all paths* solution applies the abstract transfer functions separately along each path through the program and joins only at the end,

$$\text{MOP} = \bigsqcup_{\pi \text{ a path}} f_\pi^\#(d),$$

while the *minimal fixpoint* solution is what Section 9.3 computes, joining at every merge point as it goes. Always $\text{MOP} \sqsubseteq \text{MFP}$, by monotonicity; the example shows the inequality can be strict. When every transfer function *is* a join-morphism — a *distributive* analysis — the two coincide, by an easy induction. Many classical dataflow analyses (live variables, available expressions, reaching definitions) are distributive and therefore suffer no loss; constant propagation is the canonical analysis that is not.

Computing MOP is not an option in general: a program with a loop has infinitely many paths, and even for loop-free programs the number of paths is exponential — which is the same observation made in Section 5.5 about the number of basic blocks. The abstract fixpoint is computed instead precisely because it joins eagerly, and the price is the gap above.

Relation to Section 9.7. This is Theorem 9.9(a) being strict, and strict already at a conditional, with no loop involved. The commutation hypothesis of part (b) fails at $C = \gamma(d_1) \cup \gamma(d_2)$, where abstracting first replaces a two-element set of store-pairs by the single element $(\top, \top)$, which describes four combinations rather than two.

9.8.5 Combining with signs

A final remark. Constant propagation and sign analysis are incomparable: **Const** knows that $x = 5$ but cannot say $x \geq 0$ when x varies, and **Sign** knows $x \geq 0$ but never that $x = 5$. One can of course run both and take the pair, which is the *product* domain $D_1 \times D_2$ with componentwise order and $\gamma(d_1, d_2) = \gamma_1(d_1) \cap \gamma_2(d_2)$. This is again a Galois connection, and the construction is sound for free by Theorem 9.6; but the result is only as precise as the two analyses run independently. Making the components exchange information — a *reduced* product — requires a further operation, and is a topic in its own right.

9.9 Interval Analysis

Signs and constants are both coarse: the first cannot say $x = 5$, the second cannot say $x \geq 0$. The *interval* domain subsumes both, recording for each variable a lower and an upper bound. It is the first domain in these notes with enough precision to prove the kind of fact one actually wants — that an array index stays in range, say — and it is also the first with a defect that the machinery developed so far does not handle: it has *infinite height*, so the Kleene iteration of Section 9.3 need not terminate. Repairing this is the subject of Sections 9.9.3 and 9.9.4 below, and the repair is of independent interest, because it is how abstract interpretation is made to work on every infinite-height domain.

9.9.1 The domain

Let $\mathbb{Z}_{\pm\infty} = \mathbb{Z} \cup \{-\infty, +\infty\}$ with the evident order, and

$$\text{Int} = \{\perp\} \cup \{[a, b] : a, b \in \mathbb{Z}_{\pm\infty}, a \leq b, a \neq +\infty, b \neq -\infty\}$$

where $[a, b]$ denotes $\{z \in \mathbb{Z} : a \leq z \leq b\}$. Order Int by inclusion: $\perp \sqsubseteq I$ for all I , and $[a, b] \sqsubseteq [c, d]$ iff $c \leq a$ and $b \leq d$. Then

$$[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)], \quad [a, b] \sqcap [c, d] = \begin{cases} [\max(a, c), \min(b, d)] & \text{if that is nonempty} \\ \perp & \text{otherwise,} \end{cases}$$

with $\top = [-\infty, +\infty]$. Arbitrary joins and meets exist (take infima and suprema of the bounds in $\mathbb{Z}_{\pm\infty}$), so Int is a complete lattice. The Galois connection is

$$\gamma_1(\perp) = \emptyset, \quad \gamma_1([a, b]) = \{z : a \leq z \leq b\}, \quad \alpha_1(Z) = \begin{cases} \perp & \text{if } Z = \emptyset \\ [\inf Z, \sup Z] & \text{otherwise,} \end{cases}$$

which is in fact a Galois insertion. As before, take $D = X \rightarrow \text{Int}$ with the pointwise order and lift γ_1, α_1 pointwise; D is again a complete lattice.

Infinite height. Unlike **Sign** and **Const**, the chain

$$[0, 0] \sqsubset [0, 1] \sqsubset [0, 2] \sqsubset [0, 3] \sqsubset \dots$$

is strictly increasing and infinite. So Theorem 9.3 still describes *lfp* as the join of the Kleene chain, but that chain need not stabilize after finitely many steps, and the iteration of Section 9.3 is no longer an algorithm. Section 9.9.3 fixes this.

9.9.2 Transfer functions

Arithmetic on $\mathbb{Z}_{\pm\infty}$ is the usual extension, with $(+\infty) + c = +\infty$ and so on; the products $0 \cdot (\pm\infty)$ we define to be 0, which is the right convention here because the only intervals involved are sets of integers. Then

$$\begin{aligned} [a, b] \hat{+} [c, d] &= [a + c, b + d] \\ \hat{-}[a, b] &= [-b, -a] \quad (\text{and } I \hat{-} J = I \hat{+} (\hat{-}J)) \\ [a, b] \hat{\times} [c, d] &= [\min P, \max P], \quad P = \{ac, ad, bc, bd\} \end{aligned}$$

with $\perp$ absorbing in each case. Addition and negation are exact; multiplication is exact on the endpoints but not in general the best abstraction of the product set, for the same reason as in Section 9.8.2: it evaluates each occurrence independently, so $[-1, 1] \hat{\times} [-1, 1] = [-1, 1]$ although $\{z \cdot z : |z| \leq 1\} = \{0, 1\}$.

The filters are where intervals earn their keep, since unlike signs and constants they can exploit inequalities. For a variable x with $m(x) = [a, b]$ and a constant c ,

$$\begin{aligned} \text{filt}_{x < c}^\#(m) &= m[x \mapsto [a, b] \sqcap [-\infty, c - 1]] \\ \text{filt}_{x \leq c}^\#(m) &= m[x \mapsto [a, b] \sqcap [-\infty, c]] \\ \text{filt}_{x = c}^\#(m) &= m[x \mapsto [a, b] \sqcap [c, c]] \end{aligned}$$

and symmetrically for $>, \geq$. Tests between two variables can also be used: for $x < y$ with $m(x) = [a, b]$, $m(y) = [c, d]$, put $x \mapsto [a, b] \sqcap [-\infty, d - 1]$ and $y \mapsto [c, d] \sqcap [a + 1, +\infty]$. All of these are sound because they only intersect with a superset of the states satisfying the test. If any component becomes $\perp$, the branch is unreachable.

9.9.3 What goes wrong, and widening

Take the archetypal program

$$x := 0; \text{ while } (x < 100) \{ x := x + 1 \}; \text{ skip}$$

and write $G(Y) = [0, 0] \sqcup \llbracket x := x + 1 \rrbracket^\#(\text{filt}_{x < 100}^\#(Y))$ for the loop-head operator in the single component x . The Kleene iteration is

$$\perp, [0, 0], [0, 1], [0, 2], \dots, [0, 100], [0, 100], \dots$$

It does stabilize — after 102 steps — but only because the bound is the literal 100. Replace the guard by $x < n$ with n unknown and the chain $[0, 0] \sqsubseteq [0, 1] \sqsubseteq \dots$ never stabilizes, even though its join, $[0, +\infty]$, is a perfectly good abstract invariant. We need a way to reach the limit without walking to it.

A *widening* on a lattice D is a binary operation $\nabla : D \times D \rightarrow D$ such that

(W1) $X \sqsubseteq X \nabla Y$ and $Y \sqsubseteq X \nabla Y$ for all X, Y ; and

(W2) for every sequence $(Z_n)_{n \geq 0}$ in D , the sequence defined by $Y_0 = Z_0$ and $Y_{n+1} = Y_n \nabla Z_{n+1}$ is ultimately stationary.

(W1) says ∇ is an upper bound operator — it may be larger than the join — and (W2) is what forces termination. Note that ∇ is not required to be monotone, commutative or associative, and typical widenings are none of these.

The standard widening on intervals discards a bound as soon as it moves:

$$\perp \nabla J = J, \quad I \nabla \perp = I, \quad [a, b] \nabla [c, d] = [c < a ? -\infty : a, \quad d > b ? +\infty : b]$$

that is, the lower bound is kept if the new interval does not extend below it and sent to $-\infty$ otherwise, and dually above. (W1) is immediate. For (W2): each of the two bounds of Y_n is non-increasing, respectively non-decreasing, and once it is $\mp\infty$ it stays there; a bound can therefore change at most once after the first step, so the sequence stabilizes within three steps. On $D = X \rightarrow \text{Int}$ apply ∇ componentwise; (W2) still holds, with the bound multiplied by $|X|$.

Theorem 9.10 (Widened iteration) *Let $G : D \rightarrow D$ be monotone and ∇ a widening. Define*

$$Y_0 = \perp, \quad Y_{n+1} = Y_n \nabla G(Y_n).$$

Then the sequence is ultimately stationary, say $Y_{N+1} = Y_N$, and $G(Y_N) \sqsubseteq Y_N$. Consequently, if G is the abstract loop operator corresponding to a concrete F in the sense of Section 9.4, then $\text{lfp} F \subseteq \gamma(Y_N)$.

Proof. Stationarity is (W2), applied to the sequence $Z_0 = \perp$ and $Z_{n+1} = G(Y_n)$. At the stationary point, $Y_N \nabla G(Y_N) = Y_{N+1} = Y_N$, and by (W1) applied to the second argument, $G(Y_N) \sqsubseteq Y_N \nabla G(Y_N) = Y_N$.

For the last claim, the computation in the loop case of Theorem 9.6 shows that $F(\gamma(Y_N)) \subseteq \gamma(G(Y_N))$, and $G(Y_N) \sqsubseteq Y_N$ with γ monotone gives $\gamma(G(Y_N)) \subseteq \gamma(Y_N)$. So $\gamma(Y_N)$ is a pre-fixpoint of F , and fixpoint induction (P) gives $\text{lfp} F \subseteq \gamma(Y_N)$. ■

This is the essential point, and it is worth stating separately: *soundness never required computing the least fixpoint*. Any pre-fixpoint of F is an over-approximation of $\text{lfp} F$, by (P); widening abandons the attempt to find the least one and jumps to some pre-fixpoint, trading precision for termination. What is lost is any claim of optimality — Y_N need not be $\text{lfp} G$, and the result depends on ∇ , and in a program with several loops on the order in which they are iterated.

On the running example, with G as above:

$$\begin{aligned} Y_0 &= \perp \\ Y_1 &= \perp \nabla G(\perp) = \perp \nabla [0, 0] = [0, 0] \\ Y_2 &= [0, 0] \nabla G([0, 0]) = [0, 0] \nabla [0, 1] = [0, +\infty] \\ Y_3 &= [0, +\infty] \nabla G([0, +\infty]) = [0, +\infty] \nabla [0, 100] = [0, +\infty] = Y_2 \end{aligned}$$

Here $G([0, +\infty]) = [0, 0] \sqcup \llbracket x := x + 1 \rrbracket^\sharp([0, 99]) = [0, 0] \sqcup [1, 100] = [0, 100]$. Three steps instead of a hundred and two — and the same three steps if the guard is $x < n$. The invariant obtained is $x \in [0, +\infty]$, and at the loop exit, after the filter for $\neg(x < 100)$,

$$[0, +\infty] \sqcap [100, +\infty] = [100, +\infty].$$

Sound, and weaker than the truth: x is exactly 100 there.

9.9.4 Recovering precision: narrowing

The widened result Y_N is a pre-fixpoint, and G applied to a pre-fixpoint yields a smaller pre-fixpoint. So one can simply iterate G downwards from Y_N and stay sound at every step. The only difficulty is that this descending iteration need not terminate either, which is what the dual operation addresses.

A *narrowing* is a binary operation $\triangle$ such that

(N1) $Y \sqsubseteq X$ implies $Y \sqsubseteq X \triangle Y \sqsubseteq X$; and

(N2) for every sequence (Z_n) , the sequence $W_0 = Z_0$, $W_{n+1} = W_n \triangle Z_{n+1}$ is ultimately stationary.

The standard narrowing on intervals refines only the infinite bounds:

$$[a, b] \triangle [c, d] = [a = -\infty ? c : a, b = +\infty ? d : b]$$

with $\perp$ absorbing. (N1) holds because the result lies between the two intervals when $[c, d] \sqsubseteq [a, b]$; (N2) holds because each bound can be replaced at most once, after which it is finite and frozen.

Theorem 9.11 (Narrowed iteration) *Let G be monotone and let Z_0 satisfy $G(Z_0) \sqsubseteq Z_0$. Define $Z_{n+1} = Z_n \triangle G(Z_n)$. Then (Z_n) is decreasing and ultimately stationary, and $G(Z_n) \sqsubseteq Z_n$ for every n ; so every Z_n is sound, and in particular $\text{lfp} F \sqsubseteq \gamma(Z_n)$ for all n .*

Proof. By induction, $G(Z_n) \sqsubseteq Z_n$. This is the hypothesis for $n = 0$. Given it, (N1) with $X = Z_n$, $Y = G(Z_n)$ yields $G(Z_n) \sqsubseteq Z_{n+1} \sqsubseteq Z_n$; the right-hand inequality makes the sequence decreasing, and monotonicity of G then gives $G(Z_{n+1}) \sqsubseteq G(Z_n) \sqsubseteq Z_{n+1}$, which is the claim for $n + 1$. Stationarity is (N2). Soundness of each Z_n is as in Theorem 9.10: a pre-fixpoint of G concretizes to a pre-fixpoint of F . ■

Continuing the example from $Z_0 = Y_N = [0, +\infty]$:

$$\begin{aligned} G(Z_0) &= [0, 100], & Z_1 &= [0, +\infty] \triangle [0, 100] = [0, 100] \\ G(Z_1) &= [0, 0] \sqcup \llbracket x := x + 1 \rrbracket^\sharp([0, 99]) = [0, 0] \sqcup [1, 100] = [0, 100] \\ Z_2 &= [0, 100] \triangle [0, 100] = [0, 100] = Z_1 \end{aligned}$$

so the loop invariant is $x \in [0, 100]$, and at the exit

$$[0, 100] \sqcap [100, +\infty] = [100, 100],$$

that is, $x = 100$ exactly. The widening threw away the upper bound; the narrowing put it back. Note that the narrowing cannot recover what a widening destroys in general — here it happened to — and that it only ever improves *infinite* bounds, which is why the finite lower bound 0 was never at risk.

9.9.5 Remarks

Intervals are still non-relational. The domain records a property of each variable separately, so it cannot express $i \leq n$, and the analysis of the summation program of Section 9.6 with unknown n yields $i \in [0, +\infty]$ and no more. Relational domains — octagons, which track $\pm x \pm y \leq c$, and convex polyhedra, which track arbitrary linear inequalities — recover this at increasing cost, and both have infinite height, so both need widening.

Widening is where the engineering lives. Theorem 9.10 asks very little of ∇ , and the crude widening above is correspondingly crude. Practical analyzers use *widening with thresholds*, which jumps not to $\pm\infty$ but to the next constant appearing in the program text (here 100, giving the right answer with no narrowing at all), delay widening for the first few iterations, and apply it only at a cut set of loop heads rather than at every program point. None of this affects soundness, which Theorem 9.10 secures for any ∇ satisfying (W1) and (W2); all of it affects precision.

Where this sits relative to Sections 5–8. The interval analysis of this section discovers $x \in [0, 100]$ without being told. The Hoare proof of Section 8.5 was told $0 \leq i \leq n \wedge 2s = i(i+1)$ and verified it. The two are complementary in a practical tool: an abstract interpreter supplies cheap invariants automatically — ranges, non-nullness, simple sign facts — and the verification conditions of Sections 5 and 6 are then discharged with those invariants assumed, leaving the programmer to write only the ones no domain can find.