Program verification

Readings: Chapter 4.

In this module, we define and work with a formal proof system for proving programs correct. The motivation is similar to that of previous modules: a proof can provide confidence of correctness in a situation where exhaustive semantic checking is time-consuming or impossible.

Testing versus verification

Current practice is to gather evidence for program correctness by testing – both black-box testing (in which tests are designed independent of the code) and white-box testing (in which tests are designed based on the code).

This is analogous to checking that a propositional formula is a theorem by trying a few valuations, or to checking that a predicate formula is a theorem by constructing a few models and interpretations. Exhaustive testing is difficult even for small programs, and impossible in the case where a program can consume an unbounded amount of data.

The process of formal verification starts with the formal description of a specification for a program in some symbolic logic, following that with a proof (in some proof system) that the program meets the formal specification. If the proof system is sound, then this implies that the program meets its specification for all inputs.

The question of whether the formal specification conforms to the informal notion of what the program should do is not a technical question, but a social and organizational matter, and properly the subject of a course in software development or engineering.

Formal specification and verification can help reduce or eliminate bugs, aid in code development, maintenance and extension, and facilitate interoperability and code reuse.

While formal specification is widespread in industry, formal verification is most often applied in safety-critical situations (airplanes, cars, medical equipment, nuclear power plants). Researchers continue to try to develop systems to automate verification, and to develop programming methodologies in which proofs of correctness are produced along with programs.

A core programming language

We will describe a proof system for programs written in a simple language that contains the core features of languages such as C/C++, Java, and Pascal, though it may differ slightly in syntax from these. The program syntax is part of the proof system.

This approach was first described by Tony Hoare in 1969, and has been elaborated upon and extended since then. In contrast, the techniques of chapters 3 and 5 (which we will not discuss here) assume a fixed program or set of programs, and reason about the behaviour of the system. Both approaches are valuable, but this one is more commonly used in informal reasoning about programs (e.g. when learning about algorithms).

The core language is imperative: it is based on a sequence of commands (e.g. assignment statements). It is sequential: there is no notion of concurrency or of threads. It is transformational: running the program transforms some initial state (initial values of the program variables) to some final state.

The language contains many familiar features: integer and boolean expressions, assignment, if-then-else, while loops, and arrays. But many familiar features are missing: functions, procedures, objects, pointer-based data structures, recursion. We will sketch how to add some of these, but a full treatment is beyond the scope of this course.

As an example, consider the following program `Fac1` to compute the factorial of `x`.

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

Before describing the semantics of the language (which should be intuitively familiar to you) we will first describe it syntactically.

Expressions

Integer expressions are built up from numerals, variables, and the basic operations of unary negation, addition, subtraction, and multiplication. Grammatically:

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

Negation binds more tightly than multiplication, which binds more tightly than subtraction and addition. This permits some removal of parentheses.

Examples: `y`, `z + 5`, `x * (y + 3)`.

Boolean expressions are built up from comparison, truth constants, and our familiar connectives $\wedge$, $\vee$, and $\neg$, written in a more textual form.

$$B ::= \texttt{true} \mid \texttt{false} \mid (!B) \mid (B \,\&\, B) \mid (B \parallel B) \mid (E < E)$$

The book cheats a bit by permitting more comparison operators, notably $(E == E)$, but also $(E \mathrel{!=} E)$. These can easily be built from the above operators, but the book really should have put these operators into the grammar.

We assume the usual priorities for logical operators.

Here is the part of the grammar for commands:

$$C ::= x = E \mid C; C \mid \texttt{if}\, B\, \{C\}\, \texttt{else}\, \{C\} \mid \texttt{while}\, B\, \{C\} \mid \texttt{skip}$$

We have assignment, composition (sequencing of statements), if-then-else with a condition (Boolean expression) and two code blocks, and a while-loop with a condition and a code block. These will have the semantics with which we are all intuitively familiar from our experience with programming.

The command `skip` might be unfamiliar. The `skip` is “do nothing”—it is useful for emulating loops and conditions with empty bodies.

The textbook is a bit inconsistent in its use of semicolons, but we will reproduce its programs without attempting to correct them.

The state of a program written in this language can be represented by a mapping from the names of variables to their integer values (or a vector of such values, or a tuple).

The semantics of each of the command constructs given in the grammar can be explained in terms of the transformation of this state from what it was before the command is executed to what it is after the command is executed. Because you have all programmed in an imperative language before, we will avoid a detailed explanation of the semantics of each construct; a brief trace of the execution of the example `Fac1` given earlier will suffice.

Hoare triples

The analogy to a sequent in program verification is a Hoare triple, so named because it is made up of three components.

$$(\!|\phi|\!)\ P\ (\!|\psi|\!)$$

The second component P is a program; the first and third components are formulas of predicate logic, called the **precondition** and **postcondition** respectively. The meaning of this triple is that, if the program P is run starting in a state that satisfies ϕ, then the resulting state will satisfy ψ.

In order for the predicate logic formulas to be able to say anything reasonable about the state of the program, they must have as predicates $<$ and $=$, as functions $-$ (unary negation), $+$, $-$ (binary subtraction), and $*$ (binary multiplication).

The model $\mathcal{M}$ for such formulas interprets these functions and predicates in the standard mathematical fashion, and has as an environment σ giving values to all variables constituting the state. If $\Phi^{\mathcal{M}}(\phi)(\sigma) = T$ in this manner, we write $\sigma \models \phi$, viewing σ as a state of the program.

The formulas may use variables present in the program, plus other variables not present (and this second type are the only ones that may be quantified). We also permit $\perp$ and $\top$ (“top”) to occur.

A specification of a program P we wish to write is then a Hoare triple with P as the second component. For instance, the specification of an integer square root program might be

$$(\!|x \geq 0|\!)\, P\, (\!|y * y \leq x|\!)$$

This is a bad specification, since the program `y = 0` satisfies it. A better one, satisfied by the program below it, would be

$$(\!|x \geq 0|\!)\, P\, (\!|y * y \leq x \land \forall z(z * z \leq x \rightarrow z \leq y)\,|\!)$$

```
y = 0;
while (y*y <= x) {y = y+1}
y = y-1;
```

Program variables and logical variables

When we try to specify preconditions and postconditions for this variant `Fac2`, we run into a problem.

```
y = 1;
while (x != 0) {
  y = y * x;
  x = x - 1;
}
```

We can't have as a postcondition $y = x!$, because the value of x has changed. The solution is to make the precondition $x = x_0 \wedge x \geq 0$, and the postcondition $y = x_0!$. The variable x_0, which doesn't appear in the program, is called a **logical variable**.

We will now develop a proof system for validating Hoare triples. This will be compositional in the structure of the program P; that is, we will describe rules that construct a triple involving P, given a recursive (grammatical) definition of P, and triples that involve the pieces used to build P according to that definition.

Before we consider the rules, we have to deal with the fact that a program may not terminate, in which case the meaning of its postcondition is somewhat uncertain. We distinguish partial correctness (where the postcondition is satisfied **if** the program terminates) and total correctness (where partial correctness holds, and in addition, the program is guaranteed to terminate).

Any program that is guaranteed to loop forever will satisfy all specifications as far as partial correctness is concerned. This is a problem if we want to write down a specification and automatically generate a program that meets it. It is less of a problem if we create a program that we think honestly meets its specification, since we tend not to put deliberate infinite loops in.

Nevertheless, we sometimes write programs whose termination is not obvious. In this case, we can take a proof of partial correctness and augment it with a termination argument. For most of what we discuss here, we will talk about partial correctness. (Recall from module 05 that Turing showed that termination of programs is an undecidable property.)

Proof calculus for partial correctness

There are six rules in this proof calculus. Five of them cover the four grammatical rules for building programs: composition, assignment, if, while, and skip. The sixth rule allows us to strengthen preconditions and weaken postconditions.

They are stated in a fashion similar to the sequent transformation system in exercise 1.2.6. This naturally leads to a proof tree, since every rule results in one Hoare triple, but might require more than one in order to achieve that result. We write $\vdash_{\text{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ if we prove the Hoare triple using our system.

The rule for skip is:

$$\frac{}{(\!|\phi|\!)\ \text{skip}\ (\!|\phi|\!)}\ \text{Skip}$$

Since skip is an instruction that does nothing, it maps any precondition to the same postcondition.

The rule for composition is:

$$\frac{(\!|\phi|\!)\ C_1\ (\!|\eta|\!) \qquad (\!|\eta|\!)\ C_2\ (\!|\psi|\!)}{(\!|\phi|\!)\ C_1; C_2\ (\!|\psi|\!)}\ \text{Composition}$$

This says that in order to prove $(\!|\phi|\!)\ C_1; C_2\ (\!|\psi|\!)$, we need to find a **midcondition** η for which we can prove $(\!|\phi|\!)\ C_1\ (\!|\eta|\!)$ and $(\!|\eta|\!)\ C_2\ (\!|\psi|\!)$.

The assignment rule has no premises.

$$\frac{}{(\!|\psi[E/x]|\!)\; x = E\; (\!|\psi|\!)}\ \text{Assignment}$$

As a small example: $\vdash_{\text{par}} (\!|y+1=7|\!)\; x = y+1\; (\!|x=7|\!)$ by one application of the assignment rule.

The rule for if-statements is:

$$\frac{(\!|\phi \wedge B|\!)\, C_1\, (\!|\psi|\!) \qquad (\!|\phi \wedge \neg B|\!)\, C_2\, (\!|\psi|\!)}{(\!|\phi|\!)\ \text{if}\ B\ \{C_1\}\ \text{else}\ \{C_2\}\ (\!|\psi|\!)}\ \text{If-statement}$$

Again, it helps to think about this from bottom to top. If we want to show that with precondition ϕ, the if-statement when executed results in postcondition ψ, then we need to show that in both cases (B is true or false) this happens.

In the case where B is true, the block of code C_1 is executed, and we need to show that ψ results, but in addition to ϕ being true, we also know that B is true (and this may turn out to be crucial in proving the triple on the top-left side of the rule).

Similarly, when B is false, we use this in reasoning about C_2.

The rule for while statements is:

$$\frac{(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)}{(\!|\psi|\!)\ \text{while}\ B\ \{C\}\ (\!|\psi \wedge \neg B|\!)}\ \text{Partial-while}$$

The rule is labelled “partial” because it is the one that changes in the proof calculus for total correctness (since this is the only source of non-termination).

The formula ψ is called a loop invariant. It is a key part of informal reasoning about algorithms. Coming up with a loop invariant is a good demonstration of your understanding of what a loop is intended to achieve and how it achieves it.

The final rule adjusts preconditions and postconditions.

$$\frac{\vdash_{AR} \phi' \to \phi \qquad (\!|\phi|\!)\, C\, (\!|\psi|\!) \qquad \vdash_{AR} \psi \to \psi'}{(\!|\phi'|\!)\, C\, (\!|\psi'|\!)} \text{ Implied}$$

Here the notation $\vdash_{AR} \phi' \to \phi$ is meant to indicate that the formula $\phi' \to \phi$ is provable in predicate logic with the rules of arithmetic (PA) as axioms. This means that whenever ϕ' is true, so is ϕ; in other words, ϕ' is a stronger precondition than ϕ. Similarly, ψ' is a weaker postcondition than ψ. This rule is sometimes necessary to adjust formulas in order to match them up for applications of other rules.

Proof tableaux

Here is a small piece of the proof of `Fac1`:

$$\dfrac{\dfrac{(\!|1 = 1|\!)\ \texttt{y = 1}\ (\!|y = 1|\!)}{(\!|\top|\!)\ \texttt{y = 1}\ (\!|y = 1|\!)}\,\mathrm{i} \quad \dfrac{(\!|y = 1 \wedge 0 = 0|\!)\ \texttt{z = 0}\ (\!|y = 1 \wedge z = 0|\!)}{(\!|y = 1|\!)\ \texttt{z = 0}\ (\!|y = 1 \wedge z = 0|\!)}\,\mathrm{i}}{(\!|\top|\!)\ \texttt{y = 1; z = 0}\ (\!|y = 1 \wedge z = 0|\!)}\,\mathrm{c}$$

Clearly, a flattened representation is needed.

Since a program is the composition of single statements (some of which are compound if-statements or while-statements), a proof of a whole program will consist of many applications of the composition rule. The program itself is linear, so we can linearize the proof by interleaving logical assertions.

To prove $\vdash_{\text{par}} (\!|\phi_0|\!)\, C_1; C_2; \ldots; C_n\, (\!|\phi_n|\!)$, we compose the proofs of $\vdash_{\text{par}} (\!|\phi_i|\!)\, C_{i+1}\, (\!|\phi_{i+1}|\!)$ for $i = 0, 1, \ldots, n-1$, writing this as:

$(\!|\phi_0|\!)$

C_1 ;

$(\!|\phi_1|\!)$ justification

C_2 ;

$\vdots$

$(\!|\phi_{n-1}|\!)$justification

C_n ;

$(\!|\phi_n|\!)$ justification

When we write a program P, we typically know about the precondition $(\!|\phi_0|\!)$ and the postcondition $(\!|\phi_n|\!)$. We provide the midconditions $(\!|\phi_i|\!)$ ($i = 1, 2, \ldots n-1$) as part of the proof.

Because our programming language is heavily assignment-based, and the rule for assignment makes the precondition a modification of the postcondition:

$$\frac{}{(\!|\psi[E/x]|\!)\; x = E\; (\!|\psi|\!)}\ \text{Assignment}$$

it's easier to work backwards through the program, discovering $(\!|\phi_{n-1}|\!)$, then $(\!|\phi_{n-2}|\!)$, and so on.

So, for example, if we want to show $\vdash_{\text{par}} (\!|y = 5|\!)$ `x = y + 1` $(\!|x = 6|\!)$, we push the postcondition through the assignment statement to get:

$(\!|y + 1 = 6|\!)$

```
x = y + 1 ;
```

$(\!|x = 6|\!)$ Assignment

We know that $\vdash y = 5 \rightarrow_{AR} y + 1 = 6$, so we can use the Implied rule. Since the program part is the same above and below the line in the Implied rule, we can just write the stronger formula beforehand (and/or, if we use the other part of the Implied rule, the weaker formula afterwards).

$(\!|y = 5|\!)$

$(\!|y + 1 = 6|\!)$ Implied

```
x = y + 1 ;
```

$(\!|x = 6|\!)$ Assignment

We really should include a proof of $\vdash y = 5 \rightarrow_{AR} y + 1 = 6$ by natural deduction, but this is done using techniques we have already discussed in earlier modules, so we will omit these. Make sure in your own work that you do not inadvertently include an implication that cannot be proved.

Here is a proof of a roundabout addition program derived in a "bottom-up" fashion.

$(\!| \top |\!)$

$(\!| x + y = x + y |\!)$ Implied

```
z = x ;
```

$(\!| z + y = x + y |\!)$ Assignment

```
z = z + y ;
```

$(\!| z = x + y |\!)$ Assignment

```
u = z ;
```

$(\!| u = x + y |\!)$ Assignment

To push a postcondition ψ back through an if-statement of the form `if` (B) C_1 `else` C_2, we push it through each of the blocks C_1 and C_2, yielding:

$$(\!|\phi_1|\!)C_1(\!|\psi|\!) \qquad (\!|\phi_2|\!)C_2(\!|\psi|\!)$$

In order to make this a valid application of the If-statement rule, we have to have ϕ_1 be of the form $\phi \wedge B$ and ϕ_2 be of the form $\phi \wedge \neg B$. This is highly unlikely.

Instead, we must construct a ϕ such that $\phi \wedge B \to \phi_1$ and $\phi \wedge \neg B \to \phi_2$, then use the Implied rule to get the triples above into the form needed for the application of If-statement. Using $\phi = (B \to \phi_1) \wedge (\neg B \to \phi_2)$ will work.

Since this is going to be a common occurrence, we put it together into a derived rule, to which the textbook gives the same name of “If-Statement”.

$$\frac{(\!|\phi_1|\!)\, C_1\, (\!|\psi|\!) \qquad (\!|\phi_2|\!)\, C_2\, (\!|\psi|\!)}{(\!|(B \rightarrow \phi_1) \wedge (\neg B \rightarrow \phi_2)|\!)\ \text{if } B\ \{C_1\}\ \text{else}\ \{C_2\}\ (\!|\psi|\!)}\ \text{If-statement}$$

We can leave the precondition in its rather complicated state, or simplify it using the Implied rule. Time for an example.

The following code fragment is supposed to set `y` to the absolute value of `x`.

```
if (x > 0) {
    y = x;
} else {
    y = -x
}
```

Suppose we only need the postcondition $y \geq 0$. We can push this postcondition back through each of the code blocks for the two cases.

$$(\!|x \geq 0|\!) \quad \texttt{y = x} \quad (\!|y \geq 0|\!)$$

$$(\!|-x \geq 0|\!) \quad \texttt{y = - x} \quad (\!|y \geq 0|\!)$$

Combining $\phi_1 = x \geq 0$ and $\phi_2 = -x \geq 0$ with $B = x > 0$ in the manner of the modified If-Statement rule, we see that ϕ is the formula $(x > 0 \rightarrow x \geq 0) \wedge (\neg(x > 0) \rightarrow -x \geq 0)$.

This will be the derived precondition of the if-statement. But it's not too hard to see that this is an arithmetic tautology, and thus implied by $\top$. The next slide contains the complete proof tableau.

```
        $(\!|\top|\!)$
        $(\!|(x > 0 \to x \geq 0) \land (\neg(x > 0) \to -x \geq 0)|\!)$   Implied
if (x > 0) {
            $(\!|x \geq 0|\!)$                                         If-Statement
     y = x;
            $(\!|y \geq 0|\!)$                                         Assignment
} else {
            $(\!|-x \geq 0|\!)$                                        If-Statement
      y = - x;
            $(\!|y \geq 0|\!)$                                         Assignment
}
        $(\!|y \geq 0|\!)$                                             If-Statement
```

The process of proof for if-statements is a bit involved, but not very difficult. In contrast, the process for while statements takes a fair amount of ingenuity. To see why, look at the form of the Partial-while rule.

$$\frac{(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)}{(\!|\psi|\!)\ \text{while}\ B\ \{C\}\ (\!|\psi \wedge \neg B|\!)}\ \text{Partial-while}$$

As it stands, it only allows one to prove triples in which the precondition is a slight variation of the postcondition. This is the same situation we faced with if-statements, but the solution is not so simple. The basic idea is the same: we use the Implied rule to modify the precondition. But discovering how to do this is not just a matter of pushing the postcondition through parts of the statement.

We'd like to be able to prove something like:

$$(\!|\phi|\!)\ \ \texttt{while}\ (B)\{C\}\ \ (\!|\psi|\!)$$

In order to do this, we need to find a formula η such that $\vdash_{AR} \phi \to \eta$, $\vdash_{AR} \eta \wedge \neg B \to \psi$, and (in order to use the already-defined Partial-while rule), $\vdash_{\text{par}} (\!|\eta \wedge B|\!)\, C\, (\!|\eta|\!)$.

η is called a **loop invariant**, and you can see why: passage through the loop body does not affect its validity. It may cease to be true in the middle of the body, but it is restored by the end. Discovering a loop invariant requires really understanding what the loop is accomplishing.

A loop invariant expresses a relationship between the program variables (possibly involving logical variables as well) that is preserved by the body of the loop. In effect, the body is performing a controlled evolution of the values of the program variables.

The invariant has to be weak enough to be implied by our desired precondition (if we know what it is), and it has to be strong enough to imply (together with the knowledge $\neg B$ that the loop has terminated) our desired postcondition.

Often doing a trace of a program fragment involving a while loop will give us some indication of what the relationship expressed by the invariant might be.

As an example, consider the `Fac2` program.

```
y = 1;

while (x != 0) {

  y = y * x;

  x = x - 1;

}
```

Our next example is a simple nesting of an if-statement inside a while-loop, but it is already too big to fit onto a single slide. We will annotate it in pieces.

The code fragment finds the maximum among the first n values of an array A (indexed starting from 0). To even write this code, we have to extend our simple programming language to include arrays. We will also temporarily use a variant form of an if-statement without an “else” block.

The code only needs to read from the array A and not write to it. We will see that modifying arrays leads to further complications in our proof calculus.

Here is the program `Max`.

```
c = A[0];
i = 1;
while (i != n) {
  if (c < A[i]) then
     c = A[i];
  i = i + 1;
}
```

```
      $(\!|\top|\!)$
  c = A[0];
  i = 1;
  while (i != n) $\{$
     if (c < A[i]) then
        c = A[i];
  i = i + 1;
$\}$
      $(\!|\forall j(0 \leq j < n \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < n) \wedge (c = A[j]))|\!)$
```

```
c = A[0];
i = 1;
while (i != n) $\{$
   if (c < A[i]) then
      c = A[i];
i = i + 1;
   $(\!|\forall j(0 \leq j < i \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i) \wedge (c = A[j]))|\!)$
$\}$
   $(\!|\forall j(0 \leq j < n \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < n) \wedge (c = A[j]))|\!)$
```

```
c = A[0];
i = 1;
while (i != n) {
    if (c < A[i]) then
        c = A[i];
    $\llparenthesis \forall j(0 \leq j < i+1 \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i+1) \wedge (c = A[j])$
i = i + 1;
    $\llparenthesis \forall j(0 \leq j < i \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i) \wedge (c = A[j]))\rrparenthesis$
}
```

```
if (c < A[i]) then {
    c = A[i];
} else {
    skip
}
```

$(\!|\forall j(0 \leq j < i+1 \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i+1) \wedge (c = A[j])$

What we saw was that the loop invariant

$$\eta = \forall j(0 \leq j < i \to c \geq A[j]) \wedge \exists j((0 \leq j < i) \wedge (c = A[j]))$$

when pushed through the loop increment became

$$\eta' = \forall j(0 \leq j < i{+}1 \to c \geq A[j]) \wedge \exists j((0 \leq j < i{+}1) \wedge (c = A[j])).$$

We now have to push this through the inner if-statement, which means pushing it through each of the two blocks. The “else” block has no effect, so we get η', but the “then” block does have an effect, yielding a precondition (for that block) of ϕ_1 (to be determined).

$$(\forall j(0 \leq j < i+1 \rightarrow A[i] \geq A[j])$$

$$\qquad \wedge \exists j((0 \leq j < i+1) \wedge (A[i] = A[j])))$$

```
    c = A[i];
```

$$\llparenthesis \forall j(0 \leq j < i+1 \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i+1) \wedge (c = A[j])$$

Recall the form of the If-Statement rule we adopted:

$$\frac{(\!|\phi_1|\!)\,C_1\,(\!|\psi|\!) \qquad (\!|\phi_2|\!)\,C_2\,(\!|\psi|\!)}{(\!|(B \to \phi_1) \land (\neg B \to \phi_2)|\!)\ \text{if}\ B\,\{C_1\}\ \text{else}\ \{C_2\}\,(\!|\psi|\!)}\ \text{If-statement}$$

In our case, ψ is the modified loop invariant η', ϕ_2 is just η' (because the “else” block has no effect), and

$$\begin{aligned} \phi_1 \quad = \quad & \forall j(0 \leq j < i+1 \to A[i] \geq A[j]) \\ & \quad \land \exists j((0 \leq j < i+1) \land (A[i] = A[j])). \end{aligned}$$

Our goal is to have the loop invariant η at the top of the loop body.

Combining the modified If-Statement rule with Implied, this tells us that we need to show that

$$\eta = \forall j(0 \leq j < i \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i) \wedge (c = A[j]))$$

implies $(B \rightarrow \phi_1) \wedge (\neg B \rightarrow \eta')$.

Showing $\eta \rightarrow (\neg B \rightarrow \eta')$ is easy. The other part of this is $\eta \rightarrow (B \rightarrow \phi_1)$, or

$$\begin{aligned} &\forall j(0 \leq j < i \rightarrow c \geq A[j]) \wedge \exists j((0 \leq j < i) \wedge (c = A[j])) \\ \rightarrow &(c < A[i] \rightarrow \forall j(0 \leq j < i+1 \rightarrow A[i] \geq A[j]) \\ &\qquad \wedge \exists j((0 \leq j < i+1) \wedge (A[i] = A[j])))\text{.} \end{aligned}$$

```
 c = A[0];
 i = 1;
    $(\!|\forall j(0 \leq j < i \rightarrow c \geq A[j]) \land \exists j((0 \leq j < i) \land (c = A[j]))|\!)$
 while (i != n) {
    $(\!|\forall j(0 \leq j < i \rightarrow c \geq A[j]) \land \exists j((0 \leq j < i) \land (c = A[j]))|\!)$
    if (c < A[i]) then
       c = A[i];
 i = i + 1;
}
```

```
      $(\!|\top|\!)$
      $(\!|A[0] = A[0]|\!)$
 c = A[0];
      $(\!|c = A[0]|\!)$
      $(\!|\forall j(0 \leq j < 1 \to c \geq A[j]) \wedge \exists j((0 \leq j < 1) \wedge (c = A[j]))|\!)$
 i = 1;
      $(\!|\forall j(0 \leq j < i \to c \geq A[j]) \wedge \exists j((0 \leq j < i) \wedge (c = A[j]))|\!)$
 while (i != n) $\{$
    if (c < A[i]) then
       c = A[i];
 i = i + 1;
$\}$
```

We have succeeded, with the program `Max`, in pushing the postcondition all the way through to yield the desired precondition. Note, however, that if $n < 1$, the program will not terminate. The proof calculus for total correctness allows us to prove $\vdash_{\text{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$, that is, the program P, when started in a state making ϕ true, definitely terminates and results in a state that makes ψ true.

A little thought convinces us that the only rule which needs to change is the one for while-statements.

To prove that a given while-loop terminates, we need a **variant** – an expression whose value decreases each time the loop is executed, but which is bounded below (for convenience, by zero). Letting the expression be E, and using a logical variable E_0 to denote its value before one pass through the loop body, this gives the following rule:

$$\frac{(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\; C \;(\!|\eta \wedge 0 \leq E < E_0|\!)}{(\!|\eta \wedge 0 \leq E|\!)\;\text{while}\; B \;\{C\}\; (\!|\eta \wedge \neg B|\!)}\ \text{Total-while}$$

For `Fac1`, the variant is $x - z$; for `Fac2`, it is x; for `Max`, it is $n - i$. Variants are more complicated for the more sophisticated algorithms seen in CS 240 and CS 341.

Example proof

We present on this slide and the next one a complete example of a total correctness proof for the program `Fac2`:

```
     $(\!| x = x_0 \land 0 \leq x |\!)$
     $(\!| 1 * x! = x_0! \land 0 \leq x |\!)$
y = 1;
     $(\!| y * x! = x_0! \land 0 \leq x |\!)$
while (x != 0) {
          $(\!| y * x! = x_0! \land 0 \leq x = E_0 \land x \neq 0 |\!)$
          $(\!| y * x * (x-1)! = x_0! \land 0 < x = E_0 |\!)$
    y = y * x;
          $(\!| y * (x-1)! = x_0! \land 0 < x = E_0 |\!)$
          $(\!| y * (x-1)! = x_0! \land 0 \leq x - 1 = E_0 - 1 |\!)$
    x = x - 1;
```

$$(\!|y * x! = x_0! \wedge 0 \le x = E_0 - 1|\!)$$

$$(\!|y * x! = x_0! \wedge 0 \le x < E_0|\!)$$

$\}$

$$(\!|y * x! = x_0! \wedge 0 \le x < E_0 \wedge \neg(x \neq 0)|\!)$$

$$(\!|y = x_0!|\!)$$

Further extensions

There is still a considerable gap between our simple programming language and the languages we use in practice. But since we are reaching the limit of what we can reasonably present and understand, we will discuss only in general terms how to extend our proof calculus to handle new features.

The program `Max` only read from its array A and did not modify it. As soon as we allow assignment to array elements, we break the assignment rule. The problem is **aliasing**: the ability to refer to the same array element by more than one name. This is because array elements are referred to by the value of an integer index, and two variables may have the same integer value.

Consider the following incorrect tableau, which looks okay according to the assignment rule we have.

$$(\!|A[y] = 0|\!)$$

```
x = y;
```

$$(\!|A[y] = 0|\!)$$

```
A[x] = 1;
```

$$(\!|A[y] = 0|\!)$$

The solution is to define a notation for referring to the array A in assertions that is similar to the notation we used for lookup tables in the semantics of predicate logic, and to create a version of the assignment rule that applies specifically to array elements.

Aliasing problems also arise with object references and with linked data structures. In effect, we must be more precise about the semantics of the programming language, and augment our logical formulas with the expressivity needed to speak about the state of a program including all of its data structures.

Most high-level programming languages provide the ability to define and use functions. We can define a syntax for function definition:

```
function f(x,y) { retval = x*x + y*y; }
```

and then use the function elsewhere in a program:

`y = f(a,b)`.

If we can create a tableau $(\!|\phi|\!)B(\!|\psi|\!)$ for the body of the function `f(x,y)`, we can then say that $\phi[a/x][b/y] \to \psi[a/x][b/y][f(a,b)/retval]$ and then use this in pushing a postcondition through an assignment statement `y = f(a,b)` in the program.

Something similar is possible for procedures without return values.

Recursive procedures present problems similar to while-statements. If we try to push a postcondition through the body of a recursive procedure, we discover that we need to know something about the precondition at the point where the recursive call is made.

```
procedure p(x,y) {
    (|$\phi$|)
     ...
        (|$\phi'$|)
    p(a,b);
        (|$\psi'$|)
     ...
    (|$\psi$|)
}
```

Discovering suitable ϕ and ψ to allow the proof to go through is an art similar to discovering loop invariants.

Is program verification practical?

Given the amount of effort required to verify even the simple programs we have considered, does it make sense to try to justify the correctness of larger programs and systems?

We have presented only the beginnings of the theory. Further developments in both the theory and practice of program verification are the subject of active research. There are systems that assist in the more tedious aspects, while leaving the creative parts up to human interaction. More sophisticated methods reduce the complexity and increase the size of programs for which verification is practical.

Much of the difficulty we have encountered has been due to the heavy use of assignment, or **mutation** (changing the value to which the name of a variable refers). Verification is easier for programs written in a purely functional language (e.g. Haskell) or one where mutation plays a minor role (e.g. Scheme, ML).

Although a typical program will not be subjected to full verification, aspects of verification often play an important role in development and testing. Complex algorithms are usually explained by means of assertions and invariants placed at strategic points in the program. CS 341 and CS 466 contain examples of this.

Moving preconditions and postconditions from the level of individual statements to the level of methods, classes, or modules does not result in full verification, but can increase confidence in the ability of a module or class to provide desired functionality. The textbook refers to this as “programming by contract”. Unit testing can be seen as informal verification of assertions.

Programming languages and systems are beginning to incorporate features to facilitate programming by contract (e.g. the assert statement in Java, contracts in Eiffel and PLT Scheme).

Goals of this module

You should understand and be able to use the terminology introduced in this module: Hoare triples, preconditions, postconditions, program states, program variables, logical variables, and so on.

You should commit to memory the rules for the proof calculus for partial correctness, and be able to prove small programs correct.

You should be able to discover and justify loop invariants and variants for simple programs.