A Tutorial on Functional Program Verification

Yoonsik Cheon and Melisa Vela

TR #10-26
September 2010; revised August 2011

Keywords: correctness proof; functional verification; intended function; Cleanroom;

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Correctness proofs, formal methods; D.3.3 [*Programming Languages*] Language Constructs and Features — control structures; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — logics of programs, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

A Tutorial on Functional Program Verification

Yoonsik Cheon and Melisa Vela
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968
ycheon@utep.edu; smvelaloya@miners.utep.edu

***Abstract*—This document gives a quick tutorial introduction to functional program verification. In functional program verification, a program is viewed as a mathematical function from one program state to another, and proving its correctness is essentially comparing two mathematical functions, the function computed by the program and the specification of the program. The reader is assumed to have some programming experience and to be familiar with sets and functions.**

I. Introduction

Software is used widely in almost every aspect of our daily lives. Even the smallest devices such as cellular phones and PDAs are controlled by software. Software contains defects. Defects are introduced during software development and are often found through testing. However, studies indicate that testing can't detect more than 90% of defects; 10% of defects are never detected through testing. As stated by a famous computer scientist, testing has a fundamental flaw in that it can show the existence of a defect but not its absence. Can we do better than this? Can we detect or avoid defects even before we start testing software?

In the late 70s, Harlan Mills and his colleagues at IBM developed an approach to software development called *Cleanroom Software Engineering* [1] [2]. Its name was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication, and the method reflects the same emphasis on defect prevention rather than defect removal. Special methods are used at each stage of the software development—from requirement specification and design to implementation—to avoid errors. In particular, it uses specification and verification, where verification means proving, mathematically, that a program agrees with its specification.

Cleanroom is a lightweight, or semi-formal, method and tries to verify the correctness of a program using a technique that we call *functional program verification* [3]. The technique requires a minimal mathematical background by viewing a program as a mathematical function from one program state to another and by using equational reasoning based on sets and functions. In essence, the functional verification involves (a) calculating the function computed by code called a *code function* and (b) comparing it with the intention of the code written as a function called an *intended function* [4]. For this, the behavior of each section of code is documented, as well as the behavior of the whole program. The documented behavior is the specification to which the correctness of a program is verified.

We believe that the functional program verification technique can be effectively taught and practiced, as it requires a minimal mathematical background and reflects the way programmers reason about the correctness of a program informally. It is also our conjecture that if programmers become proficient in the functional program verification, they may be able to learn easily other verification techniques such as Hoare logic as a complementary reasoning technique.

This document provides a tutorial introduction to the techniques of functional program verification. In Section II below, we first learn that we can view a program as a mathematical function and thus can specify its behavior as a function. In Section III, we then learn how to verify the correctness of code by essentially comparing two functions. We describe techniques for verifying assignment statements, sequences of statements, and conditional statements such as an **if** statement. In Section IV, we cover loop statements such as a **while** statement. In Section V, we apply the learned techniques to verify sample code.

II. Writing specifications

The correctness of code means that code does what it is supposed to do. The question then is: how do we know what the code is supposed to do? It is the specification of the code that should tell the behavior of the code. Therefore, the first step of any verification—formal or informal—is to document the expected behavior of code as its specification. In this section, we learn how to write a specification that describes the behavior of the code to be verified.

Consider the following statement involving two variables `x` and `sum`.

```
sum = sum + x;
```

Suppose that the values of `x` and `sum` are 10 and 100, respectively, before the execution of the statement, called an *initial state*. Let us denote this initial state as $\{x \mapsto 10, y \mapsto 100\}$. What do we know about the values of the variables after the execution of the statement, called a *final state*. The values of `x` and `sum` are 10 and 110, respectively; that is, the final state is $\{x \mapsto 10, sum \mapsto 110\}$. As shown, an execution of a statement, or a section of code, produces a side-effect on a program state by changing the values of some state variables such as program variables. Thus, we can view an execution of a statement, or a section of code, as a mathematical function that, given a program state, produces a new state, where a program state is a mapping from state variables to their values. For example, the above statement can be modeled as a mathematical function, say f, that maps an initial state $\{x \mapsto 10, sum \mapsto 100\}$ to a final state $\{x \mapsto 10, sum \mapsto 110\}$, $\{x \mapsto 20, sum \mapsto 100\}$ to $\{x \mapsto 10, sum \mapsto 120\}$, and so on.[1] We call such a function a *function computed by code* or shortly a *code function*.

Exercise 1. Describe the function computed by the following code.

```
x = x + y;
y = x - y;
x = x - y;
```

How do we represent a code function? We use a *concurrent assignment*, a succinct notation to express a function by only stating changes in an input state. A concurrent assignment is written as:

$$[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$$

and states that each x_i's new value is e_i, evaluated concurrently in the initial state, i.e., the input state or the state just before executing the code. The value of a state variable that doesn't appear in the left-hand side of a concurrent assignment remains the same. For example, the code function for the statement `sum = sum + x` is $[sum := sum + x]$, and a function that swaps two variables, x and y, is written as $[x, y := y, x]$.

The function computed by code is often partial in that the code works only for some well-defined input values. We extend the concurrent assignment notation to also specify such a partial function. For example, $[n \neq 0 \rightarrow avg := sum/n]$ is a function that is defined only for a state in which n is not zero. The condition is evaluated in the initial state. We often want to specify different functions based on some conditions. For example, the following function determines the sign of the variable x.

```
[x > 0 → sign := 1
 | x < 0 → sign := -1
 | else → sign := 0]
```

The conditions are evaluated sequentially from the first to the last, and if more than one condition holds, the function defined is the one corresponding to the first condition that holds. This form of concurrent assignments is called a *conditional concurrent assignment*, and it contains any number of cases, each of which contains a logical expression and a concurrent assignment. When a section of code modifies a state variable under certain conditions, its conditional concurrent assignment includes the conditions to trace how the state variables are affected.

Exercise 2. Write a conditional concurrent assignment to describe the function computed by the following code.

```
if (n > maxSize) {
    n = maxSize;
}
avg = sum / n;
```

Frequently we would like to highlight the fact that code has no side-effect or it is undefined on certain input values by explicitly stating it. This can be done by using special symbols and keywords, such as **I** and **undefined** (see examples below).

```
[n > maxSize → n := maxSize | else → I]
[n > 0 → avg := sum/n | else → undefined]
```

The first function above states that when n is less than or equal to `maxSize`, it is an identity function, denoted by **I**; that is, the code doesn't have any side-effect. The second function explicitly states that it is undefined when n is not positive. As shown, the **undefined** keyword is used to indicate that the result under a certain condition is not defined.

We use the concurrent assignment notation not only to document the actual function computed by a section of code but also to describe our intention for the code called an *intended function*. An intended function is a specification for a program or a section of code. It is a statement of the behavior that we expect from the code. As a specification, it only deals with the behavior that can be observed from outside. When we intend for a part of a program to produce a particular computation, we will describe that computation in terms of a function from the current state to a new state. For example, the intended function below states that variables x and y are initialized to 0 and 1, respectively.

```
[x, y := 0, 1]
 x = 0;
 y = 1;
```

When we write an intended function for a piece of code, we are saying specifically that the code must compute that function when the code is executed in any state in the domain of the function. When this is the case, we say that the combination comprised of the intended function and the code is *correct*, or that the code is *correct with respect to* the intended function; correctness means that code agrees with its intended function. In general, an intended function may be correctly implemented by many different code sequences, as shown below.

[1]The function f is defined as $f(\{x \mapsto X, sum \mapsto S\}) = \{x \mapsto X, y \mapsto S + X\}$.

```
[x, y := 0, 1]
 y = 1;
 x = 0;

[x, y := 0, 1]
 y = 1;
 x = y;
 x = x - y;
```

An intended function is not concerned about the values of variables in intermediate states. when there are more than one statement. It specifies only the final result of the computation. Also, an intended function doesn't care about the order in which variables are mutated when multiple variables are involved.

Frequently, the final value of a variable is expressed in terms of its initial value. In the intended function below, for example, the new value of `sum` is its initial value plus the summation of all elements of an array `a` starting at index `i`; recall that a variable appearing in the right hand side of a concurrent assignment refers to its initial value.

```
[sum, i := sum + Σ_{j=i}^{a.length−1} a[j], anything]
 while (i < a.length) {
   sum += a[i];
   i++;
 }
```

The keyword **anything** indicates that we don't care about the final value of the variable `i`. Such a variable is often called an *incidental variable*.

We use an intended function for both documentation and verification purposes (see Section III). Thus, an intended function should be written in such a way that it is easy to read and understand, it is precise and unambiguous, and it is in a form that is easy to manipulate in verification.

Exercise 3. Write intended functions for the following code.

(a)
```
sum = sum + a;
avg = sum / n;
```

(b)
```
if (a[i] == k) {
   l = i;
}
```

(c)
```
while (i < a.length) {
   if (a[i] == k) {
     l = i;
   }
   i++;
}
```

III. Verifying Correctness

Once we have a specification for code, we can check if the code conforms to its specification. *Verification* is the process of checking whether a program, or a section of code, is correct with respect to its specification. Recall that we view a program as a mathematical function and specify its behavior as an intended function. Thus, verification essentially means comparing two mathematical functions, a code function—a function computed by code—and an intended function. When code is simple, we often calculate its code function directly and compare it with the intended function of the code. When code is complicate, however, we use different techniques to verify its correctness based on the structure of the code. In this section, we learn several techniques that we can use to verify the correctness of code.

```
//@ [a is non-empty → l := index of max value in a]

  //@ [l, i := 0, 1]
  l = 0;
  int i = 1;

  /*@ [l, i := index of max value of a[l, i..n]
              anything] @*/
  while (i < a.length) {
    //@ [a[i] > a[l] → l, i := i, i+1 else → i := i+1]
      //@ [a[i] > a[l] → l := i+1 else → I]
      if (a[i] > a[l]) {
         //@ [l := i]
         l = i;
      }

      //@ [i := i+1]
      i++;
  }
```

Figure 1. Sample annotated code

A. Annotating Code

To facilitate correctness verification of a program, we annotate its code with intended functions. We write an intended function for each section of code. Figure 1 shows sample code annotated with intended functions. An annotation is written as a special kind of comments preceded by an `@` symbol. We often use indentation to indicate the region of code that an intended function annotates. The top-level intended function states that, given a non-empty array `a`, the code sets the variable `l` to the index of a maximum value in `a`. The function is partial because for an empty array the output is undefined.

The next intended function annotates the initialization code and specifies the initial values of `l` and `i`. For a control structure such as an **if** statement and a **while** statement, we specify the behavior of the whole structure as well as each of its components. Recall that we use the keyword **anything** to indicate that we don't care about the final value of a certain variable, typically a local or incidental variable. The symbol **I** denotes an identity function.

Exercise 4. Annotate the following code with intended functions.

```
c = 0;
int i = 0;
while (i < a.length) {
  if (a[i] == n)
     c++;
  i++;
}
```

B. Assignment Statement

The verification of a single assignment statement is often straightforward because it is possible to calculate the function computed by the statement, which is often identical to the intended function. For example, the following code is correct; the code function and the intended function are the same.

```
//@ [x := x + 1]
x = x + 1;
```

In general, the verification of a single statement with a code function p with respect to an intended function f involves showing that:

- The domain of p is a superset of the domain of f (dom $p \supseteq$ dom f). That is, p accepts all the values accepted by f.
- For each x in the domain of f, p maps x to the same value that f maps to ($p(x) = f(x)$ for $x \in \text{dom}(f)$).

The reason for the first condition is that often code does more than what its intended function states. For example, the following code is correct.

```
//@ [n > 0 → avg := sum / n]
avg = sum / n;
```

For the variable n, the intended function accepts only a positive number, but the code accepts any non-zero number.

C. Sequential Composition

How do we verify the correctness of a sequence of statements. Consider the following code involving two assignment statements.

```
//@ [n > 0 → sum, avg := sum+a, (sum+a)/n]
  sum = sum + a;
  avg = sum / n;
```

Since we view a statement as a mathematical function, executing one statement after another is the same as applying one mathematical function after another. That is, a sequential composition of two statements is viewed as a composition of two functions. Therefore, verification of the above code requires proving that the composition of p_1 and p_2 is correct with respect to the intended function, where p_1 and p_2 are code functions of the first and second statements, respectively. Here's a proof.

$$
\begin{array}{ll}
 & [sum := sum + a]; \\
 & [n \neq 0 \rightarrow avg := sum/n] \\
\equiv & [n \neq 0 \rightarrow sum, avg := sum + a, (sum + a)/n] \\
\sqsubseteq & [n > 0 \rightarrow sum, avg := sum + a, (sum + a)/n]
\end{array}
$$

In the proof, the notation $f_1 \sqsubseteq f_2$ means that f_1 is correct with respect to f_2, where f_1 and f_2 are code or intended functions; it is also said that f_1 is a *refinement* of f_2.

It is often error prone to calculate a function computed by a sequence of statements, especially when the statements change several variables. A *trace table* can be used to show all the state changes made by a sequence of statements [4, Chapter 3]. It is similar to executing statements symbolically. As an example, consider the following statements.

```
x = x + 1;
y = 2 * x;
z = x * y;
x = x + 1;
x = 3 * x;
```

What is the function computed by the code? Let us calculate it using a trace table. A trace table contains one row per statement and a column for the statements plus one column for each state variable that is changed or affected by any of the statements (see below).

statement	x	y	z
x = x + 1	$x+1$		
y = 2 * x		$2(x+1)$	
z = x * y			$(x+1)*2(x+1)$
x = x + 1	$x+2$		
x = 3 * x		$3(x+2)$	

A trace table simulates the execution of a statement by recording the effect of the execution on the state variables. When a value is assigned to a variable, for example, its new value is computed symbolically using the values available in the current state and is recorded on the column for that variable. For example, the execution of the second statement results in adding $2(x+1)$ to the y column because $2*x$ is assigned to y and the current value of x is $x+1$, the most recent entry in the x column. In summary, for every row in the table, there is a statement and the change made to state variables by that statement. Upon completion of a trace table, we collect the most recent entry of each variable column to produce a function computed by the statements. The above trace table produce the following function.

$$[x, y, z := x{+}2, 3(x{+}2), 2x^2{+}4x{+}2]$$

Exercise 5. Use a trace table to calculate the function computed by the following code.

```
rate = 0.5;
years++;
interest = balance * rate / 100;
balance = balance + interest;
```

It is possible to verify the correctness of a sequential composition of statements in a modular way. Consider the following skeletal code annotated with intended functions.

```
//@ [f0]
  //@ [f1]
  S1;
  //@ [f2]
  S2;
```

Its verification involves discharging the following three proof obligations.

- Composition of f_1 and f_2 is correct with respect to f_0 ($f_1; f_2 \sqsubseteq f_0$).
- S_1 is correct with respect to f_1 ($S_1 \sqsubseteq f_1$)
- S_2 is correct with respect to f_2 ($S_2 \sqsubseteq f_2$)

Note that the first obligation—the verification of f_0—is stated in terms of intended functions f_1 and f_2, not in terms of code S_1 and S_2. The intended function of a component is used in place of its code for the verification. Such verification is *modular* in that it is valid for any component code S_i as long as S_i is correct with respect to an intended function f_i. It also allows us to perform the verification of a component separately from the verification of the whole code. In short, we can use the intended function of a component when verifying the whole code.

D. Conditional Statement

For a conditional statement such as an **if** statement, we can use two different techniques to verify its correctness. As before, we can calculate its code function and compare it with its intended function. For this, a variation of a trace table called a *conditional trace table* can be used. A conditional trace table is similar to the trace table except that it has an additional column for conditions. Essentially, a new trace table is created for each case in the condition. As an example, let us calculate the function computed by the following code.

```
p = a * r;
if (a < b)
   b = b - a;
else
   b = b - p;
```

As before we create a trace table (see below). However, the difference is that when we encounter a conditional statement such as an **if** statement, we create a new trace table for each branch of the statement by first copying the existing rows and then proceeding to the computation of that branch. We also fill the condition column with the condition for that branch.

statement	condition	p	b
p = a * r		$a * r$	
if (a < b)	$a < b$		
b = b - a			$b - a$
p = a * r		$a * r$	
if (a < b)	$a \geq b$		
b = b - p			$b - (a * r)$

As before, we collect the most recent entry of each variable column to produce one function per table. The conditions in the condition column are also collected and conjoined to produce a conditional concurrent assignment. We combine all the conditional concurrent assignments to produce the function computed by the code. For example, the above table produce the following function.

```
[a < b → p, b := a*r, b-a
 | a ≥ b → p, b := a*r, b-(a*r)]
```

The second technique for the verification of a conditional statement, which is preferred in most cases, is to perform a case analysis based on the condition. Consider the following skeletal code with an intended function.

```
//@ [f]
if (B)
  S1;
else
  S2;
```

If the condition B holds, S_1 is executed; the **if** statement is equivalent to S_1. Otherwise, S_2 is executed; the **if** statement is equivalent to S_2. Therefore, we have the following two proof obligations.

- When B holds, S_1 is correct with respect to f ($B \Rightarrow S_1 \sqsubseteq f$).
- When B doesn't hold, S_2 is correct with respect to f ($\neg B \Rightarrow S_2 \sqsubseteq f$).

As an example, let us verify the following code.

```
//@ [z ≠ 0 → r := |x-y|/z]
if (x > y)
  r = (x-y)/z;
else
  r = (y-x)/z;
```

Its verification is straightforward. When $x > y$ holds, we have the following because $x - y = |x - y|$:

$$[z \neq 0 \rightarrow r := (x-y)/z] \equiv [z \neq 0 \rightarrow r := |x-y|/z]$$

Similarly, when $x > y$ doesn't hold, we have the following because $y - x = |x - y|$:

$$[z \neq 0 \rightarrow r := (y-x)/z] \equiv [z \neq 0 \rightarrow r := |x-y|/z]$$

Exercise 6. Derive proof obligations for an **if** statement without an **else** part. That is, what do you have to prove for the verification of the following skeletal code (Caution: you also need to consider the case when the condition doesn't hold).

```
//@ [f]
if (B)
   S;
```

Exercise 7. Write an intended function for the following code and prove the correctness of the code with respect to the intended function.

```
if (n > maxSize)
   n = maxSize;
sum = sum + a;
avg = sum / n;
```

IV. Verifying Iterations

Verification of code containing iterative statements such as a **while** statement is more involved. It is because there is no known algorithm to calculate the function computed by such an iterative or loop statement. In this section, we learn a technique for verifying the correctness of code consisting of iterative statements.

A. Proof Obligations

Consider the following **while** statement annotated with an intended function.

```
//@ [f]
while (B)
  S
```

We know from the operational meanings of the **while** statement that the above code is equivalent to the following code.

```
//@ [f]
if (B) {
  S
  while (B)
    S
}
```

Suppose that the original code is correct—that is, the **while** statement is correct with respect to the intended function f. This means that f and the **while** statement are equivalent. Then, for verification purposes, we may use an intended function in place of its code and replace the the **while** statement with f (see below).

```
//@ [f]
if (B) {
  S
  [f]
}
```

The rewritten code now contains only an **if** statement and a sequence statement, and we already know how to verify these statements. From this code, we can develop the following proof obligations for the **while** statement.

- If B doesn't hold, the identity function is correct with respect to f, i.e., ($\neg B \Rightarrow \mathbf{I} \sqsubseteq f$), where the symbol $\Rightarrow$ denotes logical implication and **I** denotes the identity function. If B is false, the whole statement is equivalent to an empty statement, here represented by the identity function.
- If B holds, S followed by f is correct with respect to f, i.e., ($B \Rightarrow S; f \sqsubseteq f$).

As can be guessed, we use an induction to prove the correctness of a **while** statement. The intended function is an induction hypothesis. The first obligation is a basis step, and the second is an induction step. In addition we also have to prove the termination of the loop.

B. An Example

As an example, let us prove the following code from Section II on page 3.

```
/*@ f1: [sum, i := sum + Σ_{j=i}^{L} a[j], anything]
        where L = a.length - 1 @*/
 while (i < a.length) {
   //@ f2: [sum, i := sum + a[i], i+1]
     sum += a[i];
     i++;
 }
```

Its verification requires us to discharge the following three proof obligations

1) Termination of the loop
2) Basis step: $\neg(i < a.length) \Rightarrow I \sqsubseteq f_1$
3) Induction step: $i < a.length \Rightarrow f_2; f_1 \sqsubseteq f_1$ and the correctness of f_2 and its code

How do we verify the termination of a loop? We find an expression called a *loop variant* whose value is consistently increased or decreased on each iteration of the loop. When it reaches a certain value, it makes the loop condition false, thus terminating the loop. We find a loop variant $a.length - i$. As stated in f_2, the value of i increases by 1 on each iteration of the loop. Thus, the loop variant decreases by 1 on each iteration, eventually terminating the loop when it becomes 0.

Let us next prove the correctness of the loop inductively. We first prove the basis step: $\neg(i < a.length) \Rightarrow I \sqsubseteq f_1$. When i is not less than $a.length$, we have the following, where the question mark (?) denotes an arbitrary value that we don't care about and L is $a.length - 1$.

$$\begin{aligned} f_1 &\equiv [sum, i := sum + \Sigma_{j=i}^{L} a[j], ?] \\ &\equiv [sum, i := sum + 0, ?] \\ &\equiv [sum, i := sum, ?] \\ &\sqsupseteq [sum, i := sum, i] = I \end{aligned}$$

We next prove the induction step: $i < a.length \Rightarrow f_2; f_1 \sqsubseteq f_1$. When i is less than $a.length$, we have the following.

$$\begin{aligned} f_2; f_1 &\equiv [sum, i := sum + a[i], i+1]; \\ &\quad\ [sum, i := sum + \Sigma_{j=i}^{L} a[j], ?] \\ &\equiv [sum, i := sum + a[i] + \Sigma_{j=i+1}^{L} a[j], ?] \\ &\equiv [sum, i := sum + \Sigma_{j=i}^{L} a[j], ?] \\ &\equiv f_1 \end{aligned}$$

This concludes the proof of the loop in terms of the intended function of the loop body. Lastly we need to prove the correctness of the loop body. This proof is trivial, as the intended function f_2 is identical to the function computed by the loop body.

Exercise 8. Prove the termination of the following loop.

```
while (low <= high) {
  int mid = (low + high) / 2;
  if (a[mid] < x)
     low = mid + 1;
  else if (a[mid] > x)
     high = mid - 1;
  else
     high = low - 1;
}
```

Exercise 9. In the above proof of the induction step, we introduced an intended function (f_2) for the loop body. Prove the induction step directly using the code of the body without introducing an intended function for it (Hint: use a trace table). Compare the two approaches stating their implications.

C. Initialized Loops

A loop is seldom used by itself but is preceded by an initialization. An initialization and a loop together compute something useful. A loop usually computes a function more general than one that is needed. However, it can be specialized by providing initial values. For example, the **while** statement in the previous section doesn't compute the sum of all elements of an array. However, given an index i and an accumulator sum, it adds all elements of an array a starting at index i to the accumulator sum. By initializing both i and sum to 0, we can make the code to compute the sum of an array a.

In general, the use of a loop statement such as a **while** statement has the following structure, where f_0, f_1, f_2, and f_3 are intended functions for whole code, initialization code, loop, and loop body, respectively.

```
//@ [f0]
  //@ [f1]
  S1

  //@ [f2]
  while (B) {
    //@ [f3]
    S2
  }
```

Its verification requires discharging the following proof obligations.

1) $f_1; f_2 \sqsubseteq f_0$.
2) $S_1 \sqsubseteq f_1$.
3) **while** (B) $S_2 \sqsubseteq f_2$, which requires the following subproofs.
   a) Termination of the loop.
   b) Basis step: $\neg B \Rightarrow I \sqsubseteq f_2$.
   c) Induction step: $B \Rightarrow f_3; f_2 \sqsubseteq f_2$ and $S_2 \sqsubseteq f_3$.

Often, the most difficult part of the verification of code containing a loop statement is to formulate the intended function of the loop in isolation. Recall that it is an induction hypothesis. If the hypothesis is wrong, the verification will surely fail. There are some heuristics for formulating intended functions for loops in isolation, e.g., looking at sequences of values that one wants the variables to get as a loop iterates [4, Chapter 4]. But, always remember that a loop doesn't do a computation but it completes it. An initialization determines where the computation starts, and the combination of start and completion—initialization and loop—does the job.

Exercise 10. Write intended functions for the following **while** loops in isolation.

(a)
```
while (i < a.length) {
    if (a[i] > 0) {
        sum += s[i];
    }
    i++;
}
```

(b)
```
while (n > 1) {
    n = n - 2;
}
```

Exercise 11. Prove the correctness of the following code.

```
//@[ r := n!]
  r = 1;
  int i = n;
  while (i > 1) {
    r = r * i;
    i--;
  }
```

```
int longestPlateau(int[] a) {
  int l = 0;
  int i = 0;
  while (i < a.length) {
    int cl = 0;
    int v = a[i];
    while (i < a.length && v == a[i]) {
      cl++;
      i++;
    }
    if (cl > l) {
      l = cl;
    }
  }
  return l;
}
```

Figure 2. Determining the length of the longest plateau

V. EXAMPLE VERIFICATION

In this section, we will verify a small program using the techniques that we learned in previous sections. Figure 2 shows code that, given an integer array, determines the length of the longest plateau; a plateau is an array section of equal values. For example, an array $\{1, 1, 2, 2, 2, 3\}$ has three plateaus of lengths 2, 3, and 1, respectively. The first step of verification is to document the code by

```
/*@ [result := LLP(a)] where
     LLP(a) = length of the longest plateau of a. @*/
int longestPlateau(int[] a) {
  //@ [l, i := 0, 0]
  int l = 0;
  int i = 0;

  //@ [l, i := max(l,LLP(a[i..])), anything]
  while (i < a.length) {
    /*@ [l, i := max(l, L), i + L] where
         L = length of the next plateau of a[i..]. @*/

      //@ [cl, v := 0, a[i]]
      int cl = 0;
      int v = a[i];

      /*@ [cl, i := cl + N, i + N] where
           N = num of next elems in a[i..] equal to v. @*/
      while (i < a.length && v == a[i]) {
        cl++;
        i++;
      }

      //@ [cl > l → l := cl | else → I]
      if (cl > l) {
        l = cl;
      }
  }

  //@ [result := l]
  return l;
}
```

Figure 3. Annotated code

writing intended functions for its expected behavior. Figure 3 shows an annotated version of the code. The main sections of the code are documented with their intended functions along with the method specification, the top-level intended function. In the first annotation, the pseudo variable `result` denotes the return value of the method. Below we verify the annotated code in a top-down fashion.

We first verify that the method specification is correctly implemented by the method body, considering only the main components of the method body (see below).

```
/*@ f0:[result := LLP(a)] where
     LLP(a) = length of the longest plateau of a. @*/
int longestPlateau(int[] a) {
  //@ f1:[l, i := 0, 0]
  int l = 0;
  int i = 0;

  //@ f2:[l, i := max(l,LLP(a[i..])), anything]
  while (i < a.length) { ... }

  //@ f3:[result := l]
  return l;
}
```

We follow a stepwise refinement in our proof. At this step, we need to prove the following.

1) $f_1; f_2; f_3 \sqsubseteq f_0$. The method body correctly implements the method specification.
2) Correctness of f_1, f_2, and f_3 and their code. Each intended function is correctly implemented or refined by its code.

The verifications of f_1 and f_3 are trivial. For the verification of f_3, we can treat a `return` statement as an assignment statement that assigns a value to a pseudo variable `result`; recall that we use `result` to denote the return value of a method. We will prove the refinement of f_2 in the next step. For the proof of the first obligation, we can build a trace table shown below.

statement	result	l	i
f_1		0	0
f_2		LLP(a[0..])	?
f_3	LLP(a[0..])		

Note that f_2 sets l to $\max(0, LLP(a[i..]))$, which is simplified to $LLP(a[0..])$ because i is 0; as stated in the annotation, LLP denotes the length of the longest plateau in a given array. From the above trace table, we get the following code function for the method body.

$$[result, l, i := LLP(a[0..]), LLP(a[0..], ?]$$

If we ignore local variables l and i, we have:

$$\begin{aligned} & [result := LLP(a[0..])] \\ \equiv\ & [result := LLP(a)] \\ \equiv\ & f_0 \end{aligned}$$

This concludes our proof at the top level, and we next proof the refinement of f_2 shown below.

```
//@ f2:[l, i := max(l,LLP(a[i..])), anything]
while (i < a.length) {
  /*@ f4: [l, i := max(l, L), i + L] where
       L = length of the next plateau of a[i..]. @*/
}
```

By applying the proof rules for a **while** statement that we learned in Section IV, we have the following proof obligations.

1) Termination of the loop.
2) Basis step: $\neg(i < a.length) \Rightarrow I \sqsubseteq f_2$.
3) Induction step: $i < a.length \Rightarrow f_4; f_2 \sqsubseteq f_2$.

For the termination proof, we find a loop variant $a.length - i$ and note that i increases by L on each iteration of the loop, where L is the length of the next plateau. For the basis step, we have the following proof; $LLP(a[i..])$ is undefined ($\perp$) when $\neg(i < a.length)$ holds.

$$\begin{aligned} f_2 &\equiv [l, i := \max(l, LLP(a[i..])), ?] \\ &\equiv [l, i := \max(l, \perp), ?] \\ &\equiv [l, i := l, ?] \\ &\sqsupseteq [l, i := l, i] \\ &\equiv I \end{aligned}$$

For the induction step, we have the following when $i < a.length$ holds.

$$
\begin{aligned}
f_4; f_2 \quad &\equiv \quad [l, i := \max(l, L), i + L]; \\
&\qquad [l, i := \max(l, LLP(a[i..])), ?] \\
&\equiv \quad [l, i := \max(\max(l, L), LLP(a[i + L..])), ?] \\
&\equiv \quad [l, i := \max(l, L, LLP(a[i + L..])), ?] \\
&\equiv \quad [l, i := \max(l, \max(L, LLP(a[i + L..]))), ?] \\
&\equiv \quad [l, i := \max(l, LLP(a[i..])), ?] \\
&\equiv \quad f_2
\end{aligned}
$$

We next prove the correctness of the loop body, of which code is shown below.

```
/*@ f4:[l, i := max(l, L), i + L] where
     L = length of the next plateau of a[i..]. @*/

  //@ f5: [cl, v := 0, a[i]]
  int cl = 0;
  int v = a[i];

  /*@ f6:[cl, i := cl + N, i + N] where
       N = num of next elems in a[i..] equal to v. @*/
  while (i < a.length && v == a[i]) {
    cl++;
    i++;
  }

  //@ f7:[cl > l → l := cl | else → I]
  if (cl > l) {
    l = cl;
  }
```

We have the following obligations for its proof.

1) $f_5; f_6; f_7 \sqsubseteq f_4$.
2) Refinements of f_5, f_6, and f_7. The proofs of f_5 and f_7 are straightforward, and we will prove the correctness of f_6 and its code in an exercise at the end of this section.

For the proof of the first obligation, we construct the following conditional trace table.

statement	condition	l	cl	v	i
f_5			0	$a[i]$	
f_6			N		$i + N$
if (...)	$N > l$				
f_7		N			
f_5			0	$a[i]$	
f_6			N		$i + N$
if (...)	$N \leq l$				
f_7					

Note that f_6 assigns $cl + N$ to cl, which is simplified to N because cl's current value is 0 assigned by f_5; N is the number of next elements equal to v. Similarly, the **if** condition, $cl > l$, is also written as $N > l$ because cl's current value is now N. The above trace table produces the following function for $f_5; f_6; f_7$ which is a refinement of f_4 (see below).

$$
\begin{aligned}
f_5; f_6; f_7 \quad &\equiv \quad [N > l \rightarrow l, cl, v, i := N, N, a[i], i + N \\
&\qquad \mid \textbf{else} \rightarrow cl, v, i := N, a[i], i + N] \\
&\sqsubseteq \quad [N > l \rightarrow l, i := N, i + N \\
&\qquad \mid \textbf{else} \rightarrow i := i + N] \\
&\equiv \quad [l, i := \max(l, N), i + N] \\
&\equiv \quad [l, i := \max(l, L), i + L] \text{ (because } N = L) \\
&\equiv \quad f_4
\end{aligned}
$$

In the second step we ignored local variables such as cl and v for the refinement. Also note the equivalence of L and N, the length of the next plateau starting at index i and the length of consecutive elements equal to $a[i]$ starting at index i.

Exercise 12. Prove the correctness of the refinement of f_6 above (Hint: use the proof obligations for the **while** statement).

Exercise 13. Write code that, given a sorted integer array and a value, checks if the value is contained in the array using a binary search algorithm. If the value is contained in the array, its index is returned; otherwise, -1 is returned. Prove the correctness of the code.

Exercise 14. Write code implementing quicksort and prove its correctness. The quicksort algorithm employs a divide and conquer strategy to divide a list into two sub-lists, and the steps are: (1) pick an element, called a *pivot*, from the list, (2) reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way), and (3) recursively sort the sub-list of lesser elements and the sub-list of greater elements. The base case of the recursion is a list of size zero or one.

VI. Summary

In this tutorial we introduced a functional program verification. We explained the technique by applying it to representative control structures of modern programming languages, such as assignment statements, sequential compositions, **if** statements, and **while** statements. For the proof for code involving other control structures and abstraction mechanisms we ask the readers to refer to references such as the book by Stavely [4]. For a more formal treatment of the topic including formal proof rules, refer to [3].

Acknowledgment

This work was supported in part by NSF grant DUE-0837567. Thanks to Carmen Avila, Steve Roach, and Cesar Yeep for comments.

References

[1] H. D. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, Sep. 1987.

[2] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering, Technology and Process*, ser. The SEI Series in Software Engineering. Addison-Wesley, 1999.

[3] Y. Cheon, "Functional specification and verification of object-oriented programs," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 10-23, Aug. 2010.

[4] A. M. Stavely, *Toward Zero Defect Programming*. Addison-Wesley, 1999.