

Programming Theory Basics

Yannis Kassios

January 14, 2009

In this course, there are three basic principles related to programming and specifications:

- (a) A specification is a binary expression (typically one that talks about the initial and final values of the program variables).
- (b) A program is a special kind of specification (one that can be executed).
- (c) Refinement (i.e. a specification satisfies another one) is expressed by implication $S \Rightarrow P$. Program correctness (i.e. a program A satisfies its specification B) is just refinement ($A \Rightarrow B$).

The hardest part to grasp is (b): a program is a special kind of specification. In other words, a program is a binary expression. In particular, our programming theory equates a program with a binary expression that describes what happens if we execute that program. Now, a real world program does not look at all like a binary expression. In it, there are constructs like assignments, sequential compositions, conditionals and loops. To achieve (b), our theory has to provide definitions for all these constructs. In fact, our theory defines programming notations as abbreviations for logical expressions.

Assignment

For example, assignment $x := E$ is defined as an abbreviation:

$$x := E \quad = \quad x' = E \wedge y' = y \wedge z' = z \wedge \dots$$

where x, y, z, \dots are all the program variables. The definition says that $x := E$ changes the value of x , leaving everything else alone. The important point is that, to treat assignment in our theory, we have to expand it into its binary equivalent. For example, suppose that our program variables are x and y and they are both integer. Suppose that our specification says “increase x ”. Formally, the specification is $x' > x$, i.e. the final value of x is greater than its initial value. How about assigning $x+2$ to x ? This seems to be a valid way to refine this specification. The refinement we need to prove is:

$$x := x+2 \quad \Rightarrow \quad x' > x$$

To prove this, we have to realize that $x := x+2$ is an abbreviation for a binary expression. To prove the implication, we have to expand the assignment, getting rid of the abbreviation. Because we said that our variables are x and y , the expansion is:

$$x' = x+2 \wedge \underline{y' = y}$$

Notice the underlined part. Even though y does not appear in the abbreviation $x := x+2$, it appears in the unabbreviated binary expression. The refinement proof goes as follows:

$$\begin{aligned}
 & (x := x+2) \Rightarrow x' > x && \text{expand assignment} \\
 = & x' = x+2 \wedge y' = y \Rightarrow \underline{x' > x} && \text{use context } x' = x+2 \text{ in the underlined expression} \\
 = & x' = x+2 \wedge y' = y \Rightarrow x+2 > x && \text{arithmetic} \\
 = & x' = x+2 \wedge y' = y \Rightarrow \top && \text{base} \\
 = & \top
 \end{aligned}$$

Now, you might think that $y'=y$ is not really that important. In this particular example, it isn't. But in general it is very important. You rely on the fact that the assignment changes the value of only one variable all the time when you are writing your programs (if your programming language supports pointers, then at least you hope that it is true). You will rely on it in your refinement formal proofs later on in this course too. For the time being, let us look at an example that does rely on it. Suppose that the program variables are x, y . We would like to see the exact precondition under which $x:=x+1$ refines $y'>2$. As you might guess, the precondition should be $y>2$. The reason is that $x:=x+1$ does not change y , so to ensure that y is greater than 2 in the final state, it must be true in the initial state. Our informal reasoning relies on the fact that $x:=x+1$ does not change y . So must our formal reasoning. Now for the formal proof. Remember that the formula for the exact precondition is: $\forall \sigma'. P \Leftarrow S$. In this example, our state is x, y , the specification S is $y'>2$ and the specification P is $x:=x+1$. Let us put them all together:

$$\begin{aligned}
 & \forall x', y'. y'>2 \Leftarrow (x:=x+1) && \text{expand assignment} \\
 = & \forall x', y'. y'>2 \Leftarrow x'=x+1 \wedge y'=y && \text{one-point law twice} \\
 & && \text{i.e. substitute } x+1 \text{ for } x' \text{ and } y \text{ for } y' \text{ in } y'>2 \\
 = & y>2
 \end{aligned}$$

The common pitfalls regarding assignment have to do with not understanding that assignment stands for a binary expression:

- Some people go freely from $x:=E$ to $x'=E$ or from $x'=E$ to $x:=E$ disregarding all other program variables.
- Some people confuse $=$ with $:=$. I have seen people write things like $x':=E$ which is completely wrong. I have also seen things like **if** $x:=1$ **then** ... **else** ... **fi** which is not wrong grammatically (remember, $x:=1$ is a binary expression!) but it is probably not what was intended. Remember: The assignment operator $:=$ and the equality operator $=$ are different. Make sure you understand the differences between them.

The *ok* Program

You find it in most languages as the empty statement. It is a program that performs no change to the state. In our theory, it is an abbreviation of $x'=x \wedge y'=y \wedge z'=z \wedge \dots$ where x, y, z, \dots are the program variables. Everything said about assignment above is also useful for the treatment of *ok*. Notice that $x:=x$ is equal to *ok*.

Sequential Composition

In most languages sequential composition is denoted by semicolon. In our language, it is denoted by a dot and it connects not just programs, but specifications in general. Our theory defines it again as an abbreviation: when you see a sequential composition $P.Q$ you replace it with an existential quantification:

$$\exists x'', y'', \dots \cdot (\text{substitute } x', y', \dots \text{ with } x'', y'' \text{ in } P) \wedge (\text{substitute } x, y, \dots \text{ with } x'', y'' \text{ in } Q)$$

As with assignment, to treat sequential composition, we should replace it with the binary expression it stands for. For example, in integer program variables x, y :

$$\begin{array}{ll}
 & x' > x+y. y' > x+y & \text{definition of sequential composition} \\
 = & \exists x'', y''. x'' > x+y \wedge y' > x''+y'' & \text{generalization: replace } y'' \text{ with } y'-x''-1 \\
 \Leftarrow & \exists x''. x'' > x+y \wedge y' > x''+y'-x''-1 & \text{simplify and identity} \\
 = & \exists x''. x'' > x+y & \text{generalization: replace } x'' \text{ with } x+y+1 \\
 \Leftarrow & x+y+1 > x+y & \text{simplify} \\
 = & \top &
 \end{array}$$

This means that

$$x' > x+y. y' > x+y$$

leaves the final values of variables x and y completely arbitrary; they could have any integer values.

If one or both of the operands of sequential composition are abbreviations, then we have to expand them first before expanding the sequential composition. Remember that the substitutions in the definition of sequential composition work on the binary expressions to its left and right operands, not their abbreviations. So the following examples don't make sense:

$$\begin{array}{l}
 x:=x+1. y:=y+1 = \exists x'', y''. (x:=x+1) \wedge (y'':=y''+1) \text{ WRONG!} \\
 ok. x'>y = \exists x'', y''. ok \wedge x'>y'' \text{ WRONG!}
 \end{array}$$

The correct way to expand these definitions (left as an exercise) gives respectively:

$$x' = x+1 \wedge y' = y+1$$

and

$$x'>y$$

Another mistake that I see quite often is to replace sequential composition with conjunction and vice versa. This often appears in interesting combinations with the common mistakes about assignments that I mentioned above. For example, it is not rare to see things like:

$$x:=x+2. y:=x+1 = x' = x+2 \wedge y' = x+1 \text{ WRONG!}$$

Notice in this example that the final value of y should be $x+3$ and not $x+1$. The final value of x is accidentally correct.

See the solution to Exercise 121 in pages 38-39 of the book for more common traps regarding sequential composition.

Substitution Law

The Substitution Law plays a special role in our theory. We will be using it a lot, so it is a good idea to learn how to apply it well. The reason the Substitution Law is important is that the situation in which it is useful (a sequence of assignments followed by a specification) happens all the time. The Substitution Law applies when we have an assignment connected with a specification by sequential composition

$$x:=E. P$$

It does not apply in other cases. It does not apply for example in:

Key Points

- In our theory, programs and specifications are seen as binary expressions and the common programming operators are defined as abbreviations for binary operators. The assignment, the empty statement and the sequential composition are examples of operators that are defined as abbreviations for binary expressions.
- Before using binary theory in our programs, we must expand their programming constructs into their equivalent binary expressions.
- The assignment $x := E$ is not equal to $x' = E$. Their difference is that the assignment also ensures that any programming variables other than x retain their values.
- When expanding a sequential composition $P.Q$, we have to make sure that P and Q are already expanded into binary expressions.
- Sequential composition and conjunction are not the same. Sequential composition talks about several events that happen in sequence. Conjunction ensures that all its operands are true and does not say anything about sequencing.
- The Substitution Law is a very useful tool for expanding programs. It allows us to do two expansions simultaneously.
- The Substitution Law applies when we have an assignment followed by a specification, like this:

$$x := E. P$$
 It does not apply if there is no assignment, as in

$$x' = E. P$$
 or no sequential composition as in

$$x' = E \wedge P$$
- When we have two or more assignments in sequence, it is better to start applying the Substitution Law from right to left.