SRC Technical Note
1998 - 004

March 11, 1998

Substitution: Syntactic versus Semantic

Leslie Lamport

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Substitution: Syntactic versus Semantic

Leslie Lamport

March 11, 1998

Abstract

A formalism with quantifiers permits two kinds of substitution: syntactic substitution that allows the capture of bound variables and semantic substitution that does not. When quantification is explicit, all substitution can be made semantic. When quantification is implicit, as in some formalisms used to reason about programs, both types of substitution are needed.

Consider the following definitions:

$$x \;\stackrel{\Delta}{=}\; r * \cos\theta \qquad F \;\stackrel{\Delta}{=}\; \exists\,\theta : x \neq \tan\theta \tag{1}$$

What does F equal? It should equal the result of substituting $r * \cos\theta$ for x in $\exists\,\theta : x \neq \tan\theta$. Naive substitution makes F equal to $\exists\,\theta : r*\cos\theta \neq \tan\theta$, which is how most readers would probably interpret (1). However, naive substitution can lead to problems. Naively substituting $\tan\theta$ for x in the formula $\exists\,\theta : x \neq \tan\theta$, which is valid for any x, yields the invalid formula $\exists\,\theta : \tan\theta \neq \tan\theta$. Validity is lost because the free variable θ is "captured" by the quantifier $\exists\,\theta$. Logicians therefore define substitution so it renames bound variables, when necessary, to prevent the capture of variables. Under this kind of substitution, (1) defines F to equal $\exists\,\phi : r * \cos\theta \neq \tan\phi$. I refer to naive substitution as *uniform* substitution, and I call the logician's definition *contextual* substitution.

Substitution in predicate logic is well understood. An easy way to avoid confusion is to use the following rule: a symbol may not be used as a bound variable if it already has a meaning. The definition of F in (1) violates this rule because θ already has a meaning—otherwise, the definition of x would be meaningless. Instead of (1), we can write

$$x(\theta) \;\stackrel{\Delta}{=}\; r * \cos\theta \qquad F \;\stackrel{\Delta}{=}\; \exists\,\theta : x(\theta) \neq \tan\theta$$

assuming now that θ does not already have a meaning. For predicate logic, the rule guarantees that uniform and contextual substitution are equivalent.

The distinction between uniform and contextual substitution cannot be eliminated so easily in all formalisms. Uniform substitution is defined by letting the result of substituting in $o(e_1, \ldots, e_n)$, for any operator o, equal $o(\widehat{e_1}, \ldots, \widehat{e_n})$, where $\widehat{e_i}$ is the result of substituting in e_i. Thus by definition, uniform substitution distributes over the formalism's operators. If a formalism also has a definition of contextual substitution, then the two will be equivalent iff contextual substitution distributes over all operators of the formalism. If we consider $\exists\,\theta$ to be an operator, we can say that the two types of substitution differ in predicate logic because contextual substitution does not distribute over $\exists\,\theta$. The formula $\overline{\exists\,\theta : P}$ need not be equivalent to $\exists\,\theta\ :\ \overline{P}$, where overbar ($\overline{\ }$) denotes some specific contextual substitution.

If a formalism for reasoning about programs has a definition of contextual substitution, then contextual substitution is likely to differ from uniform substitution. In particular, if the formalism has a semicolon (;) operator that corresponds to the semicolon of ordinary programming languages, then $\overline{S;\ T}$ need not equal $\overline{S};\ \overline{T}$ for formulas S and T. More precisely, I will show that contextual substitution does not distribute over semicolon in a formalism in which $x\ := x+1;\ x\ := x+1$ is equivalent to $x\ := x+2$, where $x\ := \ldots$ denotes the formula corresponding to the assignment statement.

What does it mean to substitute an expression like $r * \cos\theta$ for x in $x\ := x+1$, and why should we care? Substitution arises when implementing (or refining) one program with another. If the specification of a program is that it satisfy a postcondition S, then an implementation in which x is refined by $r * \cos\theta$ is correct iff it satisfies the postcondition $\overline{S}$, where the substitution is $x \leftarrow r * \cos\theta$ [1]. If S is a formula that represents a program, then implementing S under a refinement means implementing $\overline{S}$.

To see how such substitution is performed, consider a program with two variables x and y whose values represent the cartesian coordinates of a point in a plane. We can obtain an equivalent program with variables r and θ whose values represent polar coordinates by performing the substitution $x \leftarrow r * \cos\theta,\ y \leftarrow r * \sin\theta$. To compute the formula $\overline{x := x+1}$ obtained from $x\ := x+1$ by this substitution, we can write $x\ := x+1$ as the relation $(x' = x+1) \wedge (y' = y)$ between the old and new (primed) values of the variables, and substitute to obtain

$$(r' * \cos\theta' = r * \cos\theta + 1) \wedge (r' * \sin\theta' = r * \sin\theta)$$

Solving for r' and θ' in terms of r and θ then allows us to write $\overline{x := x+1}$ as a multiple assignment of the form $r, \theta\ := \ldots$. (When $\overline{y} = 0$ and $\overline{x} = -1$, this will be a nondeterministic assignment that sets r to 0 and θ to any value in its range.)

This particular substitution does distribute over semicolon. It is easy to show that the substitution $x \leftarrow z,\ y \leftarrow z$ does not. However, I will construct a more plausible example for which $\overline{x \ := x+1};\ \overline{x \ := x+1}$ is not equivalent to $\overline{x := x+2}$. The example is the same as the preceding one, except r and θ are hyperbolic coordinates. The substitution is $x \leftarrow r * \cosh\theta$, $y \leftarrow r * \sinh\theta$, where r and θ are real numbers. Since $(\cosh\theta)^2 \geq (\sinh\theta)^2$ if θ is real, $|\overline{x}| \geq |\overline{y}|$ for all x and y. (Hyperbolic coordinates can represent only points whose cartesian coordinates satisfy $|x| \geq |y|$.) When we compute $\overline{x \ := x+1}$, we obtain solutions for r' and θ' iff $|\overline{x}+1| \geq |\overline{y}|$. The formula $\overline{x \ := x+1}$ therefore is undefined when $|\overline{x}+1| < |\overline{y}|$.[1] In particular, $\overline{x \ := x+1}$ is undefined if $\overline{x} = -1$ and $\overline{y} = 1$. Hence, $\overline{x \ := x+1};\ \overline{x \ := x+1}$ is also undefined in this case. However, a similar calculation shows that $\overline{x := x+2}$ is undefined iff $|\overline{x}+2| < |\overline{y}|$, so it is defined when $\overline{x} = -1$ and $\overline{y} = 1$. Therefore, $\overline{x \ := x+1};\ \overline{x \ := x+1}$ is not equivalent to $\overline{x := x+2}$, so contextual substitution does not distribute over semicolon if $x \ := x+1;\ x \ := x+1$ is equivalent to $x \ := x+2$.

Contextual substitution does not distribute over semicolon because semicolon involves an implicit quantification over the intermediate values of variables, and free variables are captured by the implicit quantifiers. Programming logics typically have operators with implicit quantification—for example, the wp (weakest precondition) and sp (strongest postcondition) operators—and substitution does not distribute over them.

Substitution arises when proving that one program or system specification implements another. It does not occur in the standard theories of program correctness in which one proves that a program satisfies a property, not that one program implements another. In reasoning about concurrent systems, one does prove that one system specification implements another. As observed in [2, Section 8.3.3], substitution does not distribute over the ENABLED operator of TLA, nor over the weak and strong fairness operators WF and SF defined in terms of it. The same problem should arise in any method in which liveness properties are specified as fairness conditions on actions. Although such fairness conditions are often used in describing systems, TLA appears to be the only specification method employing them that has been sufficiently well formalized so the problem is evident.

The rule given above for making contextual and uniform substitution the same in predicate logic does not work when the quantifiers are implicit. There does not even seem to be any common notation to distinguish the two. In the definitions

$$\begin{array}{lllllll} x & \triangleq & r * \cosh\theta & \qquad & Twice(A) & \triangleq & A;\ A \\ y & \triangleq & r * \sinh\theta & \qquad & B & \triangleq & Twice(x \ := x+1) \end{array} \qquad (2)$$

[1]More precisely, it represents a statement whose execution is undefined when $|\overline{x}+1| < |\overline{y}|$. Depending on the formalism, executing a statement when it is undefined might mean that the program waits, that execution aborts, or that the program is illegal.

does B equal $\overline{x\ := x+1};\ \overline{x\ := x+1}$ or $\overline{x\ := x+1;\ x\ := x+1}$? The first interpretation, based on uniform substitution, is the more natural one. If we choose this interpretation, then we must introduce some additional notation for contextual substitution.

Should substitution be uniform or contextual? The answer is yes. Both types of substitution are needed. We want to derive new theorems from existing ones by substitution, and we can do this only with contextual substitution. In theory, contextual substitution should suffice; in practice it does not. We build a complex formula from simple pieces through a sequence of definitions. As (1) and (2) indicate, it is much easier to see what we are defining when definitions are expanded by uniform substitution. If uniform and contextual substitution are not equivalent, a practical formalism should provide both.

Acknowledgments Gordon Plotkin and Peter Hancock provided helpful comments on an earlier version. The terms *uniform* and *contextual* substitution are based on suggestions by David Gries and Fred Schneider.

References

[1] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[2] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.