Substitution and Unification

CS 440: Programming Languages and Translators
Lecture 18, Mon Apr 1

Substitution

- Substitution is the syntactic operation of replacing variables by expressions. More generically, substitution takes some sort of syntactic object (expressions / terms and logical formulas are the most common in computer science) and replaces occurrences of some variable(s) by some expression(s).
- For example, in Haskell, when we define \( p(x) = x^2 + 1 \) and then say \( p(y+1) = (y+1)^2 + 1 \), we are substituting \( y+1 \) for \( x \) in the definition of \( p \). One use of the notion of substitution is when we talk about referential transparency: \( p(y+1) \) and \( (y+1)^2 + 1 \) can be substituted for each other without changing the value of whatever expression they appear in.

Notation:\(^1\) \( t[x_1 \mapsto t_1] \) ("term \( t \) with \( x_1 \) replaced by term \( t_1 \)" or "\( t \) with \( t_1 \) for \( x_1 \" is a textual copy of \( t \) where occurrences of the variable \( x_1 \) are replaced by \( t_1 \). In general a collection of substitutions is written \( [x_1 \mapsto t_1, x_2 \mapsto t_2, ..., x_n \mapsto t_n] \). (The \( x \)'s must be unique.) We say \( x \mapsto t \) is a binding of \( x \) to \( t \) and we say \( x \) and \( t \) and bound to each other.

Example: \( (x^2 + 1)[x \mapsto y+1] \) is \( (y+1)^2 + 1 \). (Note we've added parentheses around the inserted \( y+1 \), to maintain the structure of the expression as something\(^2 + 1 \).)

On the board, substitution operations are usually written in postfix with iterated substitutions read left-to-right. Names like \( \sigma \) and \( \tau \) are common, so if \( \sigma \) is the substitution \( [x \mapsto 3] \), then \( t \sigma \) is the result of applying \( \sigma \) to \( t \).

Notation: The \( \equiv \) and \( \neq \) relationships are for textual equality. E.g., (2 + 2 = 4) evaluates to true, but 2 + 2 \( \neq \) 4. (The right-hand side, 4, is a constant; the left-hand side 2 + 2, is an infix operation.) On the other hand, we should expect \( (x+x)[x \mapsto 2] \equiv 2+2 \) because substitution is a syntactic operation.

Definition (substitution). Let \( t \) be a term and \( \sigma \) be the substitution \( [x_1 \mapsto t_1, x_2 \mapsto t_2, ..., x_n \mapsto t_n] \), then definition of \( t \sigma \) is by induction on the structure of terms:

- \( c \sigma \equiv c \), where \( c \) is a constant. Example: \( 17 \sigma \equiv 17 \) (for all \( \sigma \)).
- \( x \sigma \equiv (t_k) \) if \( x \equiv \) some \( x_k \), otherwise \( x \sigma \equiv x \). In the result, the parentheses around \( t_k \) can be omitted if they're redundant. Example: \( x [x \mapsto 17] \equiv 17 \), but \( y [x \mapsto 17] \equiv y \).
- \( (f(t_1, ..., t_n)) \sigma \equiv (f \sigma)(t_1 \sigma, ..., t_n \sigma) \). Example: \( f(x, y) [x \mapsto 17] \equiv f(17, y) \).
  - Depending on the application, we might omit \( f \) from the substitution and just use \( f(t_1 \sigma, ..., t_n \sigma) \). It depends on whether we want to be able to substitute functions or not.
  - We can extend this definition to infix operations: \( (x + y) [x \mapsto 17] \equiv (+)(x, y) \equiv (+)(17, y) \equiv 17 + y \).
- The simplest substitution is the empty one, \( \emptyset \), which contains no bindings.

---

\(^1\) The notation \( t_k[t_2/x] \) is also very popular.
• For all terms $t$, we have $t \equiv t \emptyset$. That is, applying $\emptyset$ to $t$ yields $t$ back, so $\emptyset$ is the "identity" substitution, the one which makes no change.

• This definition of substitution is actually pretty restricted. In general, people look at substituting into logical predicates. The problem becomes more complicated when you have quantified variables.

• (For example, compare $(\exists x \ (f(x) > y))[x \mapsto y-2]$ and $(\exists x \ (f(x) > y))[y \mapsto x+2]$. In the first one, $x$ is the quantified variable – should we substitute for it? In second substitution, substituting for $y$ seems okay in general, but should we really copy an $x$ into the body of the existential?)

### Unification

Unification has to do with locating equalities between symbolic terms.

• For example, if we have a term $f(X,2)$ and a term $f(5,Y)$, then if $X \equiv 5$ and $Y \equiv 3$, then the two terms are syntactically $\equiv$ because applying the substitution $[X \mapsto 5, Y \mapsto 3]$ to both terms yields $f(5,3)$.

• We say $[X \mapsto 5, Y \mapsto 3]$ is a solution to the equation $f(X,3) \equiv f(5,Y)$.

• Unification is the process of solving syntactic equations between terms by finding one substitution of variables to terms that, when applied to both sides of the equation, produces terms that are $\equiv$.

• Unification is used in typechecking (in compilers) and in automated reasoning.

• We’ll see Prolog soon, but basically, executing Prolog program involves solving a collection of logical terms through unification. Trivial example: If owns(Fred, Spot) is true, then a solution to the question "Is there an $X$ such that owns(Fred, $X$) is true?" is "$X = \text{Spot}"."

• The most basic kind of unification, first-order unification, deals with languages that include variables, constants, and function terms $\text{function}(\text{term}_1, \text{term}_2, \ldots \text{term}_n)$.

• We’re not using variables as function names (no "higher-order unification" of "$X(1) \equiv f(1)$ if $X \equiv f$ ").

• We’re also not looking at the possible semantics of functions (no "$\text{plus}(X, Y) \equiv \text{plus}(Y, X)$")

• Any given equation might have multiple solutions ($f(X) \equiv f(Y)$ works if $X \equiv Y \equiv 0, X \equiv Y \equiv 1$, etc).

• We’re interested in the most general unifier (mgu) for equations ($f(X) \equiv f(Y)$ works if $X \equiv Y$).

• Definition (Generalization and Specialization of Terms): Given terms $t_1, t_2$ and substitution $\sigma$, if $t_1 \equiv t_2 \sigma$, then $t_1$ is more general than $t_2$; equivalently, $t_2$ is more special than $t_1$.

• If a substitution $\sigma$ just maps variables to variables, then $t$ and $t \sigma$ are renamings of each other, and each is simultaneously more general and more special than the other. (Basically, renamings don’t affect generality.)

• However, not all maps of variables to variables are renaming operations. For example, $[X \mapsto Z, Y \mapsto Z]$ maps $f(X, Y)$ to $f(Z, Z)$, so $f(X, Y)$ is more general than $f(Z, Z)$, but there isn’t any substitution that takes $f(Z, Z)$ to $f(X, Y)$, so $f(X, Y)$ is strictly more general than $f(Z, Z)$ (and $f(Z, Z)$ is strictly more specific than $f(X, Y)$).

• Definition (Generalization and Specialization of Substitutions): Given substitutions $\sigma$ and $\tau$, if $t \sigma$ is more general than $t \tau$ for all terms $t$, then $\sigma$ is more general than $\tau$; equivalently $\tau$ is more special than $\sigma$. We also say $\sigma$ subsumes $\tau$ and $\tau$ is subsumed by $\sigma$.

• Example: $[X \mapsto f(Y)]$ is more general than $[X \mapsto f(g(Y))]$. 
• **Example:** Let $t_1 \equiv t [X \mapsto Z]$ and $t_2 \equiv t [X \mapsto Z, Y \mapsto Z]$. We know $t_1$ is more general than $t_2$ because there's a substitution that takes $t_1$ to $t_2$, namely $[Y \mapsto Z]$. This is true for all terms $t$, so $[X \mapsto Z]$ is more general than $[X \mapsto Z, Y \mapsto Z]$.

**Equations and Unification Problems**

• **An unification equation** is a pair of terms $(t, u)$; it represents the statement "$t$ and $u$ can be unified", or "$t \sigma \equiv u \sigma$" where $\sigma$ is unknown. We can also just write $t \equiv u$

• A **unification problem** is a set of equations $\{ t_1 \equiv u_1, t_2 \equiv u_2, \ldots, t_n \equiv u_n \}$.

• A **solution** to this unification problem is a substitution $\sigma$ that simultaneously solves the equations $t_1 \sigma \equiv u_1 \sigma$, $t_2 \sigma \equiv u_2 \sigma$, $\ldots$, $t_n \sigma \equiv u_n \sigma$. We also say $\sigma$ is a unifier of for the problem.

• We say $\sigma$ is a **most general unifier (mgu)** for the problem if it is a more general substitution than all other unifiers for the problem.

  - It turns out that $\sigma$ is unique "up to renaming": Any other most general unifier $\sigma'$ is a renaming of $\sigma$.

**A Unification Algorithm**

• Let problem $P = \{ t_1 \equiv u_1, t_2 \equiv u_2, \ldots, t_n \equiv u_n \}$. I.e., $P$ is a set of unification equations. If, say, $t_1$ is just a variable $X$, then the problem $t_1 \equiv u_1$ is $X \equiv u_1$, which is easy to solve with $[X \mapsto u_1]$.

• The goal of the algorithm is to take $P$ and transform it into an equivalent problem where all the equations are of this solved form: $\{ X_1 \equiv s_1, X_2 \equiv s_2, \ldots, X_m \equiv s_m \}$ (recall, the $X$'s are variables; the $s$'s are terms).

• The algorithm works by repeatedly taking an equation $t \equiv u$ and using it to produce a simpler problem. This might involve directly solving $(t_1, u_1)$, or throwing it away if it's useless, or breaking it down into a number of smaller equations to solve. Of course, the algorithm can also fail, if there exists no solution.

  let $S = \emptyset$  // $S$ is the set of solution equations
  while $P$ has an equation $(t \equiv u)$
    . let $P'$ be $P$ less the member $(t \equiv u)$
    . if $t \equiv u$ then skip // we're throwing out the equation $(t \equiv t)$ as unnecessary
    . else if $u \equiv$ some variable $X$, then add the equation $(X \equiv t)$ to $P'$
    . else if $t$ is a constant $c$ then
      . if $u \equiv$ the same constant $c$ then skip else fail
    . else if $t \equiv f(v_1, v_2, \ldots, v_n)$ then
      . . if $u$ is a function call $g(w_1, w_2, \ldots, w_m)$ where $f \equiv g$ and $m = n$
      . . then add the equations $(v_1 \equiv w_1), (v_2 \equiv w_2), \ldots, (v_n \equiv w_n)$ to $P'$
      . . else fail
    . else $(t$ is a variable $X)$
      . . apply the substitution $[X \mapsto u]$ to the equations of $P'$
      . . add $(X \equiv u)$ to the solution set $S$
    . continue the loop with $P'$ for $P$
  end