Substitution and Unification, v.2

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:**\( t[x_1 \mapsto t_1] \) ("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 substitution binding of \( x \) to \( t \) and we say \( x \) and \( t \) and bound to each other. (From here on, we'll abbreviate "substitution binding" to just "binding".)

**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 \( \text{something}^2 + 1 \).)

- 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:

- \( c \sigma = c \), where \( c \) is a constant. Example: \( 17 \sigma \equiv 17 \) (for all \( \sigma \)).
- \( x \sigma = (t_k) \) if \( x \equiv t_k \), 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: \( (x + y)[x \mapsto 17] \equiv (+(x, y) \equiv (+(17, y) \equiv 17 + y \).

---

1 The notation \( t_1[t_2/x] \) is also very popular.
The simplest substitution is the **empty** one, $\emptyset$, which contains no bindings.

- 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 $f(X, 3)$ 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 $f(X, 3) \equiv f(5, Y)$.
  - So above, $f(X, 3) [X \mapsto 5, Y \mapsto 3] \equiv f(5, 3)$ and $f(5, Y) [X \mapsto 5, Y \mapsto 3] \equiv f(5, 3)$, showing that the substitution is indeed a solution.
  - An example of an equation that has no solution is $g(X, X) \equiv g(8, z)$ because there's no value for $X$ that is $\equiv 8$ and $\equiv z$ at the same time. (Identifiers like $z$ don't get substituted for.)
  - On the other hand, $g(X, X) \equiv g(8, Y)$ does have a solution: $[X \mapsto 8, Y \mapsto 8]$.
    - The substitution $[X \mapsto 8, Y \mapsto X]$ also works, but we'll look for substitutions where the new term doesn't involve any of the variables being substituted for. (I.e., no variable on both sides of a $\mapsto$ binding in a solution substitution.)
    - This will let us get to the most concrete substituted equations more quickly.
      - E.g., if we apply $[X \mapsto 8, Y \mapsto X]$ to $g(X, X) \equiv g(8, Y)$, we get $g(8, 8) \equiv g(8, X)$.
      - We have to apply $[X \mapsto 8, Y \mapsto X]$ one more time to get to $g(8, 8) \equiv g(8, 8)$.

- 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 $\text{owns(Fred, Spot)}$ is true, then a solution to the question "Is there an $X$ such that $\text{owns(Fred, X)}$ 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 $\text{function(term}_1, \text{term}_2, \ldots \text{term}_n)$.
  - We're not using variables as function names (no "higher-order unification" of "$X(1) \equiv f(1)$ if $X \mapsto f$").
• 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: E.g., \(f(X) \equiv f(Y)\) is solved by \([X \mapsto Y, Y \mapsto 0]\) and by \([X \mapsto Y, Y \mapsto 1]\), and so on.

• We’re interested in the most general unifier (mgu) for equations (for \(f(X) \equiv f(Y)\), the most general solution is \([X \mapsto Y]\).

**Definition (Generalization and Specialization of Substitutions):** Given terms \(t_1, t_2\) and substitution \(\sigma\), if \(t_1 \equiv t_2 \; \sigma\), then \(t_2\) is more general than \(t_1\); 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 for every term \(t\), \((t \; \sigma)\) is more general than \((t \; \tau)\), then \(\sigma\) is more general than \(\tau\); equivalently \(\tau\) is more specific than \(\sigma\). We also say \(\sigma\) subsumes \(\tau\) and \(\tau\) is subsumed by \(\sigma\).

• **Example:** \([X \mapsto f(Y)]\) is more general than \([X \mapsto f(g(Z))]\).

  • We had \([Y \mapsto f(g(Y))]\) before, and technically that substitution is the same one that replaces \(Y\) by an infinitely long term: \([Y \mapsto f(g(f(g(f(g(\ldots))))))]\). To avoid that, we’d have to do an "occur check" for \(Y\) to find it substituted term \(f(g(Y))\).

• **Example:** Let \(t_1 \equiv t\; [X \mapsto Z]\) and \(t_2 \equiv t\; [X \mapsto Z, Y \mapsto Z]\), where \(t\) is an arbitrary term. We know \(t_1\) is more general than \(t_2\) because there’s a substitution that takes \(t_1\) to \(t_2\), namely \([Y \mapsto Z]\). Since \(t\) was any term, we know \([X \mapsto Z]\) 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 \(\sigma\).

\[2\) The parentheses are optional.\]
A Unification Algorithm

- Let problem \( P = \{t_1 \equiv u_1, t_2 \equiv u_2, \ldots, t_k \equiv u_k\} \). I.e., \( P \) is a set of unification equations. If, say, \( t_1 \) is just a variable \( X_1 \), then the problem \( t_1 \equiv u_1 \) is \( X_1 \equiv u_1 \), which is solved by \( X_1 \mapsto u_1 \).

- The goal of the algorithm is to take \( P \) and develop a list of substitutions \( [X_1 \mapsto s_1, X_2 \mapsto s_2, \ldots, X_m \mapsto s_m] \) that unify all the original equations. The substitution list is initially empty; as it grows, \( P \) gets smaller.

- The algorithm works by repeatedly removing an arbitrary equation \( t \equiv u \) from \( P \) 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. If there exists no solution, the algorithm fails.

\[
\begin{align*}
\text{let } S &= ∅ \quad // S \text{ is the set of solution equations} \\
\text{while } P \text{ has an equation } (t \equiv u) & \quad \text{let } P' \text{ be } P \text{ less the member } (t \equiv u) \\
. & \quad \text{if } u \equiv \text{ some variable } X \text{, then add the substitution } [X \mapsto u] \text{ to } S \\
. & \quad \text{else if } t \equiv \text{ a constant or identifier then } // \text{ recall, variables get substituted for; identifiers don't.} \\
. & \quad \quad \text{if } u \equiv \text{ the same constant or identifier, then skip else fail}\footnote{I took out the earlier check for } t \equiv u. \text{ It makes the constant / identifier check redundant, but it slows the algorithm.} \\
. & \quad \quad \text{else if } t \equiv f(v_1, v_2, \ldots, v_n) \text{ then} \\
. & \quad \quad \quad \text{if } u \equiv \text{ a function call } g(w_1, w_2, \ldots, w_m) \text{ where } f \equiv g \text{ and } m = n \\
. & \quad \quad \quad \quad \text{then add the equations } (v_1 \equiv w_1), (v_2 \equiv w_2), \ldots, (v_n \equiv w_n) \text{ to } P' \\
. & \quad \quad \quad \text{else fail} \\
. & \quad \quad \quad \text{else } (t \text{ is a variable } X \text{, so the equation is } X \equiv u) \\
. & \quad \quad \quad \quad \text{apply the substitution } [X \mapsto u] \text{ to the equations of } P' \\
. & \quad \quad \quad \quad \quad \text{I.e., take each equation } w \equiv v \text{ in } P' \text{ and replace it by } w[X \mapsto u] \equiv v[X \mapsto u] \text{ (see below).} \\
. & \quad \quad \quad \quad \quad \text{add } (X \mapsto u) \text{ to the solution set } S \\
. & \quad \quad \quad \quad \text{continue the loop with } P' \text{ for } P \\
\end{align*}
\]

- The reason for taking the equation \( X \equiv u \) and applying the substitution \( [X \mapsto u] \) to the equations of \( P' \) is that we want to make sure that \( X \) doesn't appear in the modified \( P' \).

- Recall this lets us calculate \( S \) so that we only have to apply it once to the original equations to verify that we have a solution. E.g., take \( \{g(X, X) \equiv g(8, Y)\} \); this turns into the problem \( \{X \equiv 8, X \equiv Y\} \).

- We remove \( X \equiv 8 \) from \( P \), set \( P' \) to \( \{X \equiv Y\} \) and add \( [X \equiv 8] \) to the solution, but first we apply \( [X \mapsto 8] \) to \( P' \) to turn \( \{X \equiv Y\} \) into \( \{8 \equiv Y\} \).

- Eventually our solution will be \( [Y \mapsto 8, X \mapsto 8] \); to verify it’s correct, we can apply it to the original problem \( \{g(X, X) \equiv g(8, Y)\} \) and get \( \{g(8, 8) \equiv g(8, 8)\} \).

- If we hadn’t applied \( [X \mapsto 8] \), we would have kept \( \{X \equiv Y\} \) as is, which would have led to
  - either \( [X \mapsto Y, X \mapsto 8] \), which isn’t a legal solution (it has two bindings for \( X \))
  - or \( [Y \mapsto X, X \mapsto 8] \) (if we flipped \( X \mapsto Y \) to \( Y \mapsto X \) to avoid having two bindings for \( X \)).
But parallel substitution of \([Y \mapsto X, X \mapsto 8]\) into \([g(X, X) \equiv g(8, Y)}\) yields \([g(8, 8) \equiv g(8, X)}\). We have to apply \([Y \mapsto X, X \mapsto 8]\) again to get \([g(8, 8) \equiv g(8, 8)}\).

(Note the parallel substitution \([Y \mapsto X, X \mapsto 8]\) isn't the same as the sequential substitution \([X \mapsto 8][Y \mapsto X]\).

**Examples of Unification**

Here's a table with some examples of unification problems and their solutions (if any). This isn't an exhaustive set of tests, but is a good place to start.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution, if any</th>
</tr>
</thead>
<tbody>
<tr>
<td>([X \equiv Y, X \equiv 3})</td>
<td>([Y \mapsto 3, X \mapsto 3}) (not ([Y \mapsto 3, X \mapsto Y}))</td>
</tr>
<tr>
<td>([X \equiv 1, X \equiv 3})</td>
<td>Fails: Tries to unify 1 and 3</td>
</tr>
<tr>
<td>({f(a, Y) \equiv f(X, b), c \equiv Z})</td>
<td>([Z \mapsto c, Y \mapsto b, X \mapsto a})</td>
</tr>
<tr>
<td>({f(X) \equiv g(Y)})</td>
<td>Fails: different function names</td>
</tr>
<tr>
<td>({f(X, Y) \equiv f(X)})</td>
<td>Fails: different # of function arguments</td>
</tr>
<tr>
<td>({f(f(f(a, Z), Y), X), W \equiv f(W, f(X, f(Y, f(Z, a))))}) (Found this one in Wikipedia)</td>
<td>([Z \mapsto a, Y \mapsto f(a, a), X \mapsto f(f(a, a), f(a, a)), W \mapsto f(f(f(a, a), f(a, a)), f(f(a, a), f(a, a)))])</td>
</tr>
</tbody>
</table>

Note: Since problems and solutions are sets, reordering elements doesn't make a problem or solution different. Similarly, flipping an equation \(t \equiv u\) to \(u \equiv t\) doesn't change a problem, nor does repeating an equation. For testing purposes, you should check that your program gives the same answer if equations are reordered, flipped, or duplicated, of course.