Constructive Logic (15-317), Fall 2015
Recitation 12: Unification, Substitution, and the Robot Uprising (11/16/2015)
Vincent Huang (vincenth@andrew), Anna Gommerstadt (hgommers@cs), Michael Coblenz (mcoblenz@cs)

Today's goal is to study substitution and unification. By better understanding substitution, we will be better prepared for the upcoming robot substitution. Some have called this substitution an "uprising," but this term is needlessly inflammatory.

1 Substitution Review

A substitution is a homomorphism on terms and finitely supported, i.e., a function $\sigma : Term \to Term$ on the set of terms such that:

$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma) \qquad \textit{for all function symbols } f \textit{ and terms } t_i$$
$$\sigma = id \qquad \textit{for almost all variables}$$

We represent substitutions as: $(r/x, s/y, t/z)$.

For each of the following, tell whether it is a well-defined and valid substitution. If it is valid, write what it does when applied to the term $x \wedge x(x)$.

- $(r/x, z/x)$
  Invalid; duplicate entries for x.
- $(r/x, z/r)$
  $r \wedge x(r)$
- $(r/x)(z/r)$
  $z \wedge x(z)$
- $(z/r)(r/x)$
  $r \wedge x(r)$

Now, suppose we have $\hat{l} = (l_1/x_1, \ldots, l_n/x_n)$ and $\hat{k} = (k_1/y_1, \ldots, k_m/y_n)$. What is $\hat{k}\hat{l}$?

Recall that a *substitution* is a function σ from terms to terms that satisfies

$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma) \qquad \text{for all function symbols } f \text{ and terms } t_i$$

and has a finite domain $\mathrm{dom}(\sigma) = \{x \ : \ x\sigma \neq x\}$ of variables. Recall that a *representation ℓ for a substitution* is of the form, e.g.:

$$\ell = (r_1/x_1, r_2/x_2, \ldots, r_n/x_n)$$

For any such representation ℓ of a substitution, let $\hat{\ell}$ denote the substitution belonging to that representation ℓ.

Construct a representation of the substitution that is a composition $\hat{k}\hat{\ell}$ of $\hat{\ell}$ after $\hat{k}$ from the representation k of $\hat{k}$ and the representation ℓ of $\hat{\ell}$.

The easiest solution assumes both substitution representations to have disjoint domains: $\mathrm{dom}(\hat{\ell}) \cap \mathrm{dom}(\hat{k}) = \emptyset$. If

$$\ell = (r_1/x_1, r_2/x_2, \ldots, r_n/x_n)$$
$$k = (s_1/y_1, s_2/y_2, \ldots, s_m/y_m)$$

Then their composition $\hat{k}\hat{\ell}$ alias $\hat{\ell}$ after $\hat{k}$ is represented by

$$(s_1\hat{\ell}/y_1, s_2\hat{\ell}/y_2, \ldots, s_m\hat{\ell}/y_m, r_1/x_1, r_2/x_2, \ldots, r_n/x_n)$$

Why is this right? The portion that corresponds to ℓ is already correct, since ℓ happens second: we don't need to fix it up. But the portion that corresponds to k needs to have ℓ applied afterward.

So the representation of $\hat{k}\hat{\ell}$ can be computed from the representations ℓ and k as the list:

$$(s_i\hat{\ell}/y_i, 1 \leq i \leq m) \cup \ell$$

Without the assumption $\mathrm{dom}(\hat{\ell}) \cap \mathrm{dom}(\hat{k}) = \emptyset$, the representation of $\hat{k}\hat{\ell}$ is

$$(s_i\hat{\ell}/y_i, 1 \leq i \leq m) \cup (r_i/x_i, 1 \leq i \leq n, x_i \notin \{y_1, \ldots, y_m\})$$

This is just like the above solution, except that we deal with intersecting domains by making sure that if both k and ℓ map a variable, the portion of the map corresponding to k does the job, since that substitution is defined to happen first.

Question: Some substitutions can also be applied to formulas. Define how for the cases of $\neg A$, $A \vee B$, and $\forall x.A$. If your substitution is undefined in some cases, briefly explain why.

Answer: The substitution σ applies to formulas satisfying the following conditions:

$$\begin{aligned} (\neg A)\sigma &= \neg(A\sigma) \\ (A \vee B)\sigma &= (A\sigma) \vee (B\sigma) \\ (\forall x.A)\sigma &= \forall x.(A\sigma) \qquad \text{if } x \notin \mathrm{dom}(\sigma) \cup \mathrm{cod}(\sigma) \end{aligned}$$

The condition on x is, in general, needed for correctness to avoid capture:

$(\forall x.p(y))(x/y)$ is not $\forall x.(p(y)(x/y))$ which would be $\forall x.p(x)$ instead of $p(y)$

$(\forall x.p(x))(y/x)$ is not $\forall x.(p(x)(y/x))$ which would be $\forall x.p(y)$ alias $p(y)$

More general capture-avoiding substitutions can be defined as well that rename bound quantifiers as needed on demand to avoid capture.

2 Unifiers

The substitution σ is a unifier for the terms s and t if $s\sigma = t\sigma$. Two terms s and t are called unifiable if a unifier σ exists.

Question: Why is this a good definition of unification? Why can't we have two substitutions, one for each term?

Answer: It might seem like we should allow two different substitutions for unification: one for s and one for t. Let's try it by unifying $f(x, g(y))$ and $f(h(a), g(z))$. So we want x to be $h(a)$ and y to be z, right? Sounds good so far. But what if we want to unify $f(x, y)$ and $f(g(y), g(z))$? For the left side, we pick $(g(z)/x, g(z)/y)$. For the right, we pick $()$. Now we apply the substitutions and get $f(g(z), g(y))$. But this was wrong because these two terms should not unify: there's nothing x and y can be that makes those two terms the same. This is exactly the point of unification: pick values for the variables so that the two things that unify are the same.

Question: Are some unifiers better than others?

Answer: Consider $\sigma = (h(a)/x, z/y)$, $\sigma' = (h(a)/x, z/y, b/c)$, which are unifiers for $f(x, g(y))$ and $f(h(a), g(z))$. σ' is less general than σ: it consists of σ followed by an extra, superfluous substitution.

Definition (Most-general unifier) A substitution μ is a most-general unifier for the set of terms T iff μ unifies T and for all unifiers σ for T there is a substitution σ' such that $\sigma = \mu\sigma' = \sigma' \circ \mu$.

In other words, most-general unifiers are exactly the maximal elements with respect to the order: $\sigma \prec \mu$ iff there is a σ' such that $\sigma = \sigma' \circ \mu$. That is, the "smallest" elements consist of a most-general unifier followed by a bunch of extra substitutions.

3 Forward computation

What is forward computation for? We can discuss the following situations:

- Sometimes you don't want to just know if a single fact is true; you want to know all the true facts.
- Sometimes a computation uses smaller subproblems as building blocks and it makes sense to build up your conclusions in order. Think of dynamic programming or memoization.