ConstLog: Constructive Logic

Lecture Notes on Unification

Frank Pfenning André Platzer

Carnegie Mellon University || Karlsruhe Institute of Technology

Lecture 19

1 Introduction

In this lecture we take the essential step that makes the exact choice of goal and rule instantiation explicit in the operational semantics of logic programming. This consists of describing an algorithm for a problem called *unification* which, given two terms t and s, tries to find a substitution θ for its free variables such that $t\theta = s\theta$, if such a substitution exists. Of course, one needs to precise about how θ is computed, because, as is most obvious via answer set substitutions, this determines the result of the computation. Recall that we write $t\theta$ for the result of applying the substitution θ to the term t.

These lecture notes are conceptual of prior lecture notes [Pfe06, Sch12].

2 Using Unification in Proof Search

Before we get to specifics of the algorithm, we consider how we use unification in proof search. Let us reconsider the (by now tired) example of unary addition

$$\frac{}{\mathsf{plus}(\mathsf{0}, N, N)}\ \mathsf{pz} \qquad\qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

and an atomic goal such as

$$\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), P).$$

Clearly the conclusion of the first rule does not match this goal, but the second one does. What question do we answer to arrive at this statement?

The first attempt might be: *"There is an instance of the rule such that the conclusion matches the goal."* When we say *instance* we mean here the result of substituting terms for the variables occuring in a rule, proposition, or term. We can see that this specification is not quite right: we need to instantiate the goal as well, since P must have the form $\mathsf{s}(P_1)$ for some as yet unknown P_1. The subgoal in that case would be $\mathsf{plus}(0, \mathsf{s}(\mathsf{s}(0)), P_1)$ according to the rule instantiation with 0 for M, $\mathsf{s}(\mathsf{s}(0))$ for N, and P_1 for P.

The second attempt would therefore be: *"There is an instance of the rule and an instance of the goal such that the two are equal."* This does not quite capture what we need either. For example, substituting $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$ for P in the goal and $\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))$ for P in the rule, together with the substitution for M and N from above, will also make the goal and conclusion of the rule identical, but is nonetheless wrong. The problem is that it would overcommit: using P_1 for P in the rule, on the other hand, keeps the options open. P_1 will be determined later by search and other unifications. In order to express this, we define that t_1 is *more general* than t_2 if t_1 can be instantiated to t_2.

The third attempt is therefore: *"Find the most general instance of the rule and the goal so that the conclusion of the rule is equal to the instantiated goal."* Phrased in terms of substitutions, this says: find θ_1 and θ_2 such that $P'\theta_1 = P\theta_2$, and any other common instance of P' and P is an instance of $P'\theta_1$.

In terms of the algorithm description it is more convenient if we redefine the problem slightly in this way: *"First rename the variables in the rule so that they are disjoint from the variables in the goal (standardize apart). Then find a single most general substitution θ that unifies the renamed conclusion with the goal."* Here, a unifying substitution θ is most general if any other unifying substitution is an instance of θ.

In the remainder of the lecture we will make these notions more precise and present an algorithm to compute a *most general unifier*.

Notation: In this lecture, we follow the general convention in logic that logical variables are called x, y, z while nullary function symbols are called a, b, c. Function symbols are called f, g, h while predicate symbols are called p, q, r. Recall that Prolog has the convention of starting variables with upper-case letters X, Y, Z while starting all function and predicate symbols including nullary with lower-case letters $a, b, c, f, g, h, p, q, r$.

3 Substitution

A substitution σ is a function that replaces variables by terms. When applied to a term, σ uniformly replaces all the variables it affects in the term by their replacement but leaves the term unchanged otherwise. Different notations exist in the literature. In this lecture we adopt the notation $t\sigma$ for the result of applying substitution σ to term t, instead of $\sigma(t)$.

Definition 1 (Substitution). A *substitution* is a homomorphism on terms and finitely supported, i.e., a function $\sigma : \mathrm{Term} \to \mathrm{Term}$ on the set of terms such that

$$\begin{aligned} f(t_1, \ldots, t_n)\sigma &= f(t_1\sigma, \ldots, t_n\sigma) && \text{for all function symbols } f \text{ and terms } t_i \\ \sigma &= \mathrm{id} && \text{for almost all variables} \end{aligned}$$

That is, the *domain* $\mathrm{dom}(\sigma) = \{x \ : \ x\sigma \neq x\}$ of all variables that are affected by σ is finite.

The effect of a substitution is uniquely determined already by describing its effect on the finitely many variables that it affects. A substitution σ can be represented by listing the replacements for each of the variables in $\mathrm{dom}\,\sigma$. The most common notation for the substitution σ that replaces x by r and simultaneously replaces y by s and z by t and leaves all other variables unchanged since $\mathrm{dom}(\sigma) = \{x, y, z\}$, is denoted:

$$(r/x, s/y, t/z)$$

This substitution is usually pronounced "r for x, s for y, and t for z". It is enough to specify this substitution like that, because its effect on any other term is uniquely determined by Def. 1. Of course, it does not make sense to mention a replacement for x multiple times such as in $(r/x, s/x)$ since that ill-formed representation does not give a function.

Substitutions can be extended to work also on logical formulas. For example, $p(t)\sigma = p(t\sigma)$ for all predicate symbols p. But more attention is needed with quantifiers to avoid capture of variables such as when using the above substitution on the formula $(\forall x p(x)) \wedge (\forall r q(y))$. Quantifiers do not occur during proof search in Prolog, since that is all about unifying a goal $p(t)$ with a corresponding clause $p(s)$, possibly with multiple arguments.

4 Composing Substitutions

Since substitutions are functions, they can be composed and it is easy to see that their composition is still a substitution.

Theorem 2 (Composition of substitutions)**.** *The* composition $\sigma \circ \tau$ *of substitution σ after substitution τ is a substitution. For compatibility with the postfix notation for applying substitutions, it is denoted $\tau\sigma$ so literally written as σ after τ.*

$$t(\tau\sigma) = t(\sigma \circ \tau) = (t\tau)\sigma$$

Proof: Both σ and τ only affect finitely many variables, and so will their composition.

$$x(\tau\sigma) = x(\sigma \circ \tau) = \sigma(\tau(x)) = (x\tau)\sigma$$

Since both σ and τ leave function symbols unchanged and work homomorphically on the arguments, the same condition will be satisfied when applying σ after τ.

$$\begin{aligned} f(t_1, \dots, t_n)(\tau\sigma) &= f(t_1(\tau\sigma), \dots, t_n(\tau\sigma)) \overset{\text{IH}}{=} f((t_1\tau)\sigma, \dots, (t_n\tau)\sigma) \\ &= f(t_1\tau, \dots, t_n\tau)\sigma = (f(t_1, \dots, t_n)\tau)\sigma \end{aligned}$$

$\square$

A particularly useful type of substitutions are those that are *idempotent*, i.e., $\sigma \circ \sigma = \sigma\sigma = \sigma$, which means that applying the substitution twice has the same effect as applying it only once. Substitutions in which the variable they substitute occurs in its own replacement will not be idempotent so that things change again when they are applied multiple times. For example, applying $(f(x)/x)$ twice will turn $g(x)$ into $g(f(f(x)))$ while applying it once will only give $g(f(x))$. Idempotent substitutions are easier to work with, because we do not have to pay attention how often we apply it to a term.

A representation of the composed substitution $\tau\sigma$ can be computed from the representations of the substitutions σ and τ. If $\operatorname{dom}\sigma \cap \operatorname{dom}\tau = \emptyset$, the *composed representation* is easier just by applying the subsequent substitution σ already to the replacement list of τ:

$$\tau\sigma = (t\sigma/x \ : \ t/x \in \tau) \cup \sigma$$

where $\cup$ denotes the union of the representations of a substitution. If $\operatorname{dom}\sigma$ and $\operatorname{dom}\tau$ overlap, then the replacements that σ does to variables of $\operatorname{dom}\tau$ will have to be dropped during $\cup$. Those variables x will already have been replaced by (τx) by the time σ is applied, so are already covered on the left. Fortunately, Prolog unification during proof search works on clauses that have already been renamed to use disjoint variable names for different clauses as well as different for each use of the same clause (standardize apart). Such disjointness conditions can be helpful. The set of all variables occurring in the substitution terms is the *codomain* $\operatorname{cod}(\theta) = \bigcup_{x \in \operatorname{dom}(\theta)} \operatorname{FV}(x\theta)$. By $\operatorname{FV}(t)$ we denote the set of all free variables of term t, which are all variables in t.[1]

If $\operatorname{dom}\sigma \cap \operatorname{dom}\tau = \emptyset$, the above representation of the composition $\tau\sigma$ can be obtained step by step when considering the elements of the list representing τ, where $(\cdot)$ is the empty substitution:

$$\begin{aligned}(t/x, \tau)\sigma &= (t\sigma/x, \tau\sigma)\\ (\cdot)\sigma &= \sigma\end{aligned}$$

5 Unifiers

Definition 3 (Unifier)**.** 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.

Unifiers are not unique. For example, both $\sigma_1 = (h(a)/x, z/y)$ and $\sigma_2 = (h(a)/x, y/z)$ unify

$$f(x, g(y)) \text{ and } f(h(a), g(z)) \tag{1}$$

Neither of those unifiers is inherently better than the other. In contrast, $\sigma_3 = (h(a)/x, b/y, b/z)$ also unifies (1). But σ_3 is worse than σ_1 and σ_2, because σ_3 is unnecessarily specific since

[1]Representations of compositions $\sigma\theta$ of representations σ and θ of substitutions are conceptually particularly easy when $\operatorname{dom}(\theta) \cap \operatorname{cod}(\theta) = \emptyset$ for all the relevant substitutions (which implies idempotence) and $\operatorname{dom}(\sigma) \cap (\operatorname{dom}(\theta) \cup \operatorname{cod}(\theta)) = \emptyset$, since there is no overlapping effect in that case [Pfe06].

it specializes both variable y and variable z to the nullary constant symbol b. After having applied σ_1 to unify the terms in (1), we can still obtain the result of using the more special unifier σ_3 by subsequently instantiating y and z to b.

Definition 4 (Most-general unifier)**.** A substitution μ is a *most-general unifier* for the terms s and t iff μ unifies s and t and

$$\text{for all unifiers } \sigma \text{ of } s \text{ and } t \text{ there is a substitution } \sigma' \text{ such that } \sigma = \mu\sigma' = \sigma' \circ \mu$$

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 $\sigma = \sigma' \circ \mu$.

By this definition, σ_3 is certainly not a most-general unifier for (1) but can still be represented as $\sigma_3 = (b/y, b/z) \circ \sigma_1$. Likewise, $\sigma_2 = (y/z) \circ \sigma_1$. Since that, indeed, holds for all other unifiers of (1), it turns out that σ_1 is a most-general unifier for (1).

Most-general unifiers are still not unique, because σ_2 also is a most-general unifier of (1) as well. After establishing that σ_1 is a most-general unifier for (1), which requires more thought, it is easy to see that σ_2 is a most-general unifier using $\sigma_1 = (z/y) \circ \sigma_2$. Every unifier σ already has a substitution σ' such that $\sigma = \sigma' \circ \sigma_1$ since σ_1 is a most-general unifier. Thus, every unifier σ also has a substitution $\sigma' \circ (z/y)$ such that

$$(\sigma' \circ (z/y)) \circ \sigma_2 = \sigma' \circ ((z/y) \circ \sigma_2) = \sigma' \circ \sigma_1 = \sigma$$

Yet, when we have two most-general unifiers, they cannot be too far apart either. If we have two most-general unifiers μ, μ', then we can get from μ to μ' by composing with some σ' ($\mu' = \sigma' \circ \mu$), and we can, conversely, also get from μ' to μ with some σ ($\mu = \sigma \circ \mu'$). Neither this σ nor this σ' can possibly substitute any variable by another term that isn't itself a variable. Suppose σ did replace some x by some function term $f(e)$, then f will always remain there (if x occurs in the input terms s and t at all) so the substitution σ' can never undo that effect of adding an f. It, thus, turns out that the most-general unifier is almost unique. The most-general unifier of a set of terms is unique up to variable renaming:

Lemma 5. *If μ, μ' are most-general unifiers for the terms s and t, then there is a variable renaming σ such that $\mu = \sigma \circ \mu'$. A* variable renaming *is a substitution whose only effect is to replace variables by variables, not by arbitrary terms, and that, moreover, never renames two different variables to the same variable (which could never be undone by another substitution).*

6 Unification

As usual in this class, we present the algorithm to compute a most general unifier as a judgment, via a set of inference rules. The judgment has the form $t \doteq s \mid \theta$, where we think of t and s as inputs and a most general unifier θ as the output. We read $t \doteq s \mid \theta$ as "t unifies with s under θ". In order to avoid the n-ary nature of the list of arguments, we will have an auxiliary judgment $\mathbf{t} \doteq \mathbf{s} \mid \theta$ for sequences of terms $\mathbf{t}$ and $\mathbf{s}$. Notions such as application of substitution are extended to sequences of terms in the obvious way. We use $(\cdot)$ to stand for an empty sequence of terms (as well as the empty substitution, which is a sequence of term and variable pairs). We will use boldface letters to stand for sequences of terms.

We first consider function terms and term sequences.

$$\frac{\mathbf{t} \doteq \mathbf{s} \mid \theta}{f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta} \qquad \frac{}{(\cdot) \doteq (\cdot) \mid (\cdot)} \qquad \frac{t \doteq s \mid \theta_1 \quad \mathbf{t}\theta_1 \doteq \mathbf{s}\theta_1 \mid \theta_2}{(t, \mathbf{t}) \doteq (s, \mathbf{s}) \mid \theta_1\theta_2}$$

Observe how the most-general unifier θ of $t \doteq s$ from the left premise is already applied to the sequence on the right. Second, the cases for variables.

$$\frac{}{x \doteq x \mid (\cdot)} \qquad \frac{x \notin \mathrm{FV}(t)}{x \doteq t \mid (t/x)} \qquad \frac{t = f(\mathbf{t}), x \notin \mathrm{FV}(t)}{t \doteq x \mid (t/x)}$$

The condition that $t = f(\mathbf{t})$ in the last rule (i.e., that the term t begins with a function symbol f) ensures that it does not overlap with the rule for $x \doteq t$. The condition $x \notin \mathrm{FV}(t)$ in the last two rules also ensures that they do not overlap with the rule for $x \doteq x$. The condition that $x \notin \mathrm{FV}(t)$ is necessary because, for example, the two terms x and $f(x)$ do not have unifier: no matter what, the substitution $f(x)\theta$ will always have one more occurrence of f than $x\theta$ and hence the two cannot be equal. Observe how this relates to the fact that the substitution $f(x)/x$ that we would otherwise read of from the rule for $x \doteq f(x)$ is not idempotent since its replacement of x mentions x again.

The other situations where unification fails is an equation of the form $f(\mathbf{t}) = g(\mathbf{s})$ for two different function symbols $f \neq g$, and for two sequences of terms of unequal length. The latter can happen if function symbols are overloaded at different arities, in which case failure of unification is the correct result.

Because of the so-called occurs check $x \notin \mathrm{FV}(t)$ whenever introducing a substitution t/x, the above rules will only ever create substitutions that are *idempotent*, which can be shown by a straightforward induction. Furthermore, the only place where substitutions are merged is the rule for $(t, \mathbf{t}) \doteq (s, \mathbf{s}) \mid \theta_1\theta_2$. By the occurs check, no variable in $\mathrm{dom}(\theta_1)$ will occur in its replacement. Consequently, no variable from $\mathrm{dom}(\theta_1)$ will occur in $\mathbf{t}\theta_1$ or $\mathbf{s}\theta_1$ so also not in the right premise $\mathbf{t}\theta_1 \doteq \mathbf{s}\theta_1 \mid \theta_2$ nor its unifier θ_2. Thus, the composition $\theta_1\theta_2$ in the conclusion $(t, \mathbf{t}) \doteq (s, \mathbf{s}) \mid \theta_1\theta_2$ will be for substitutions with disjoint domains $\mathrm{dom}(\theta_1) \cap \mathrm{dom}(\theta_2) = \emptyset$ and is, thus, straightforward on their representations.

7 Soundness

There are a number of properties we would like to investigate regarding the unification algorithm proposed in the previous section. In today's lecture we will look at the first property of soundness, that is, we would like to show that the substitution θ is indeed a unifier. If the judgment $t \doteq s \mid \theta$ holds, then, indeed, applying the resulting substitution θ to t gives the exact same term as applying the substitution θ to s does.

Theorem 6 (Soundness). *If $t \doteq s \mid \theta$ then $t\theta = s\theta$.*

Proof: Since the rule for $f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta$ with function symbol applications reduces unification to unification of a sequence of terms, we need to generalize the induction hypothesis to cover the auxiliary unification judgment on term sequences.

(i) If $t \doteq s \mid \theta$ then $t\theta = s\theta$.

(ii) If $\mathbf{t} \doteq \mathbf{s} \mid \theta$ then $\mathbf{t}\theta = \mathbf{s}\theta$.

The proof proceeds by mutual induction on the structure of the deduction $\mathcal{D}$ of $t \doteq s$ and $\mathcal{E}$ of $\mathbf{t} \doteq \mathbf{s}$. This means that if one judgment appears in the premiss of a rule for the other, we can apply the appropriate induction hypothesis.

In the proof below we will occasionally refer to *equality reasoning*, which refers to properties of equality in our mathematical language of discourse, not properties of the judgment $t \doteq s$. There are also some straightforward lemmas we do not bother to prove formally, such as $t(s/x) = t$ if $x \notin \mathrm{FV}(t)$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{E}\\ \mathbf{t} \doteq \mathbf{s} \mid \theta\end{array}}{f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta}$ where $t = f(\mathbf{t})$ and $s = f(\mathbf{s})$.

$\mathbf{t}\theta = \mathbf{s}\theta$ By IH(ii) on $\mathcal{E}$
$f(\mathbf{t})\theta = f(\mathbf{s})\theta$ By definition of substitution

Case: $\mathcal{D} = \dfrac{}{(\cdot) \doteq (\cdot) \mid (\cdot)}$ where $\mathbf{t} = \mathbf{s} = (\cdot)$ and $\theta = (\cdot)$.

$(\cdot)\theta = (\cdot)\theta$ By equality reasoning

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E}_2\\ t_1 \doteq s_1 \mid \theta_1 & \mathbf{t}_2\theta_1 \doteq \mathbf{s}_2\theta_1 \mid \theta_2\end{array}}{(t_1, \mathbf{t}_2) \doteq (s_1, \mathbf{s}_2) \mid \theta_1\theta_2}$ where $\mathbf{t} = (t_1, \mathbf{t}_2)$, $\mathbf{s} = (s_1, \mathbf{s}_2)$ and $\theta = \theta_1\theta_2$.

$t_1\theta_1 = s_1\theta_1$ By IH(i) on $\mathcal{D}_1$
$(t_1\theta_1)\theta_2 = (s_1\theta_1)\theta_2$ By equality reasoning
$t_1(\theta_1\theta_2) = s_1(\theta_2\theta_2)$ By substitution composition (Theorem 2)
$(\mathbf{t}_2\theta_1)\theta_2 = (\mathbf{s}_2\theta_1)\theta_2$ By IH(ii) on $\mathcal{E}_2$
$\mathbf{t}_2(\theta_1\theta_2) = \mathbf{s}_2(\theta_1\theta_2)$ By substitution composition
$(t_1, \mathbf{t}_2)(\theta_1\theta_2) = (s_1, \mathbf{s}_2)(\theta_1\theta_2)$ By defn. of substitution

Case: $\mathcal{D} = \dfrac{}{x \doteq x \mid (\cdot)}$ where $t = s = x$ and $\theta = (\cdot)$.

$x(\cdot) = x(\cdot)$ By equality reasoning

Case: $\mathcal{D} = \dfrac{x \notin \mathrm{FV}(s)}{x \doteq s \mid (s/x)}$ where $t = x$ and $\theta = (s/x)$.

$$\begin{aligned} x(s/x) &= s && \text{By defn. of substitution} \\ &= s(s/x) && \text{Since } x \notin \mathrm{FV}(s) \end{aligned}$$

Case: $\mathcal{D} = \dfrac{t = f(\mathbf{t}), x \notin \mathrm{FV}(t)}{t \doteq x \mid (t/x)}$ where $s = x$ and $\theta = (t/x)$.

$$\begin{aligned} t(t/x) &= t && \text{Since } x \notin \mathrm{FV}(t) \\ &= x(t/x) && \text{By defn. of substitution} \end{aligned}$$

□

In a sense, soundness of the unification procedure is all that a correct deduction in logic programming depends on, since that is what determines whether a rule has been matched in plausible ways by a proper unifier. So whenever $t \doteq s \mid \theta$, then θ is indeed a unifier of t and s. However, if θ is a unifier but not a most-general unifier, then proof search may be incomplete, because θ might then overcommit to an overly specific substitution for which no proof exists, while a proof would have still existed for the most-general unifier (recall Sect. 2).

Prolog, however, actually prescribes a fixed proof search strategy: first applicable clause for the left-most subgoal is used with instantiation of their most-general unifier. If the unifier θ from the judgment $t \doteq s \mid \theta$ is merely a unifier but not most general, then we might not only have led to incomplete proof search, but might have also missed the fact that the subgoals unifies with another clause at all. Thus, for Prolog, we also need to make sure $t \doteq s \mid \theta$ gives the most-general unifier and that we find out when t and s are not unifiable in the first place so that we can move on to consider the next clause instead. These questions will be considered next.

8 Completeness

Completeness of the unification algorithm states that if s and t have a unifier, then there exists a most general one according to the algorithm. We then also need to observe that the unification judgment is deterministic to see that, if interpreted as an algorithm, it will always find a most general unifier if one exists. That is, if t and s have a unifier σ, then the judgment $t \doteq s \mid \theta$ will hold for a unifier θ that is more general, i.e., $\sigma = \theta\sigma'$ for some σ'.

Theorem 7. *If $t\sigma = s\sigma$ then $t \doteq s \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'.*

Proof: As in the soundness proof, we generalize the induction hypothesis to address sequences of terms.

(i) If $t\sigma = s\sigma$ then $t \doteq s \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'.

(ii) If $\mathbf{t}\sigma = \mathbf{s}\sigma$ then $\mathbf{t} \doteq \mathbf{s} \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'.

The proof proceeds by mutual induction on the structure of $t\sigma$ and $\mathbf{t}\sigma$. We proceed by distinguishing cases for t and s, as well as $\mathbf{t}$ and $\mathbf{s}$. This structure of argument is a bit unusual: mostly, we distinguish cases of the subject of our induction, be it a deduction or a syntactic object. In the situation here it is easy to make a mistake and incorrectly attempt to apply the induction hypothesis, so you should carefully examine all appeals to the induction hypothesis below to make sure you understand why they are correct.

Case: $t = f(\mathbf{t})$. In this case we distinguish subcases for s.

Subcase: $s = f(\mathbf{s})$.

$f(\mathbf{t})\sigma = f(\mathbf{s})\sigma$	Assumption
$\mathbf{t}\sigma = \mathbf{s}\sigma$	By defn. of substitution
$\mathbf{t} \doteq \mathbf{s} \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'	By IH(ii) on $\mathbf{t}\sigma$
$f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta$ and still $\sigma = \theta\sigma'$	By rule

Subcase: $s = g(\mathbf{s})$ for $f \neq g$. This subcase is impossible:

$f(\mathbf{t})\sigma = g(\mathbf{s})\sigma$	Assumption
Contradiction	By defn. of substitution

Subcase: $s = x$.

$f(\mathbf{t})\sigma = x\sigma$	Assumption
$\sigma = (f(\mathbf{t})\sigma/x, \sigma')$ for some σ'	By defn. of subst. and reordering
$x \notin \mathrm{FV}(f(\mathbf{t}))$	Otherwise $f(\mathbf{t})\sigma \neq x\sigma$
$f(\mathbf{t}) \doteq x \mid (f(\mathbf{t})/x)$ so we let $\theta = (f(\mathbf{t})/x)$	By rule
$\sigma = (f(\mathbf{t})\sigma/x, \sigma')$	See above
$= (f(\mathbf{t})\sigma'/x, \sigma')$	Since $x \notin \mathrm{FV}(f(\mathbf{t}))$
$= (f(\mathbf{t})/x)\sigma'$	By defn. of composition
$= \theta\sigma'$	Since $\theta = (f(\mathbf{t})/x)$

Case: $t = x$. In this case we also distinguish subcases for s and proceed symmetrically to the above.

Case: $\mathbf{t} = (\cdot)$. In this case we distinguish cases for $\mathbf{s}$.

Subcase: $\mathbf{s} = (\cdot)$.

$(\cdot) \doteq (\cdot) \mid (\cdot)$	By rule
$\sigma = (\cdot)\sigma$	By defn. of composition

Subcase: $\mathbf{s} = (s_1, \mathbf{s}_2)$. This case is impossible:

$(\cdot)\sigma = (s_1, \mathbf{s}_2)\sigma$	Assumption
Contradiction	By definition of substitution

Case: $\mathbf{t} = (t_1, \mathbf{t}_2)$. Again, we distinguish two subcases.

Subcase: $\mathrm{s} = (\cdot)$. This case is impossible, like the symmetric case above.

Subcase: $\mathrm{s} = (s_1, \mathbf{s}_2)$.

$(t_1, \mathbf{t}_2)\sigma = (s_1, \mathbf{s}_2)\sigma$	Assumption
$t_1\sigma = s_1\sigma$ and	
$\mathbf{t}_2\sigma = \mathbf{s}_2\sigma$	By defn. of substitution
$t_1 \doteq s_1 \mid \theta_1$ and	
$\sigma = \theta_1\sigma_1'$ for some θ_1 and σ_1'	By IH(i) on $t_1\sigma$
$\mathbf{t}_2(\theta_1\sigma_1') = \mathbf{s}_2(\theta_1\sigma_1')$	By equality reasoning
$(\mathbf{t}_2\theta_1)\sigma_1' = (\mathbf{s}_2\theta_1)\sigma_1'$	By subst. composition (Theorem 2)
$\mathbf{t}_2\theta_1 \doteq \mathbf{s}_2\theta_1 \mid \theta_2$ and	
$\sigma_1' = \theta_2\sigma_2'$ for some θ_2 and σ_2'	By IH(ii) on $(\mathbf{t}_2\theta_1)\sigma_1'$ $(= \mathbf{t}_2\sigma)$
$(t_1, \mathbf{t}_2) \doteq (s_1, \mathbf{s}_2) \mid \theta_1\theta_2$	By rule
$\sigma = \theta_1\sigma_1' = \theta_1(\theta_2\sigma_2')$	By equality reasoning
$= (\theta_1\theta_2)\sigma_2'$	By associative composition (Theorem 2)

□

A proof by mutual induction on the structure of t and $\mathbf{t}$ would fail (see Exercise 1). An alternative way we can restate the first induction hypothesis is:

> For all r, s, t, and σ such that $r = t\sigma = s\sigma$, there exists a θ and a σ' such that $t \doteq s \mid \theta$ and $\sigma = \theta\sigma'$.

And accordingly for the second induction hypothesis. The proof then is by induction on the structure of r, although the case we distinguish still concern the structure of s and t. For the subcase $\mathrm{s} = (s_1, \mathbf{s}_2)$, it is crucial to note that $(\mathbf{t}_2\theta_1)\sigma_1'$ is a part of $(t_1, \mathbf{t}_2)(\theta_1\sigma_1')$, which is $\mathbf{t}\sigma$, to be able to appeal to the induction hypothesis for $(\mathbf{t}_2\theta_1)\sigma_1' = (\mathbf{s}_2\theta_1)\sigma_1'$.

9 Termination

From the completeness proof in the previous section we can see that the deduction of $t \doteq s \mid \theta$ is bounded by the structure of the common instance $r = t\theta = s\theta$. The induction in the completeness proof was on the structure of that common instance, so no part of the proof could have appealed to a larger instance. Since the rules furthermore have no nondeterminism and the occurs-checks in the variable/term and term/variable cases also just traverse subterms of r, it means a unifier (if it exists) can be found in time proportional to the size of r.

Unfortunately, this means that this unification algorithm is exponential in the size of t and s. For example, the only unifier for

$$g(x_0, x_1, x_2, \ldots, x_n) \doteq g(f(x_1, x_1), f(x_2, x_2), f(x_3, x_3), \ldots a)$$

has 2^n occurrences of a.

Nevertheless, it is this exponential algorithm with a small, but significant modification that is used in Prolog implementations. This modification (which makes Prolog unsound from the logical perspective!) is to omit the check $x \notin \mathrm{FV}(t)$ in the variable/term and term/variable cases and construct a circular term. This means that the variable/term case in unification is constant time, because in an implementation we just change a pointer associated with the variable to point to the term. This is of crucial importance, since unification in Prolog models parameter-passing from other languages (thinking of the predicate as a procedure), and it is not acceptable to take time proportional to the size of the argument to invoke a procedure.

This observation notwithstanding, the worst-case complexity of the algorithm in Prolog is still exponential in the size of the input terms, but it is linear in the size of the result of unification. The latter fact appears to be what rescues this algorithm in practice, together with its straightforward behavior which is important for Prolog programmers.

All of this does not tell us what happens if we pass terms to our unification algorithm that do *not* have a unifier. It is not even obvious that the given rules terminate in that case (see Exercise 2). Fortunately, in practice most non-unifiable terms result in a clash between function symbols rather quickly.

10 Historical Notes

Unification was originally developed by Robinson [Rob65] together with resolution as a proof search principle. Both of these critically influenced the early designs of Prolog, the first logic programming language. Similar computations were described before, but not studied in their own right (see [BS01] for more on the history of unification).

It is possible to improve the complexity of unification to linear in the size of the input terms if a different representation for the terms and substitutions is chosen, such as a set of multi-equations [MM76, MM82] or dag structures with parent pointers [PW78]. These and similar algorithms are important in some applications [Kni89], although in logic programming and general theorem proving, minor variants of Robinson's original algorithm are prevalent.

Most modern versions of Prolog support sound unification, either as a separate predicate `unify_with_occurs_check/2` or even as an optional part of the basic execution mechanism[2]. Given advanced compilation technology, I have been quoted figures of 10% to 15% overhead for using sound unification, but I have not found a definitive study confirming this.

Another way out is to declare that the bug is a feature, and Prolog is really a constraint programming language over rational trees, which requires a small modification of the unification algorithm to ensure termination in the presence of circular terms [Jaf84] but still avoids the occurs-check. The price to be paid is that the connection to the predicate calculus is lost, and that popular reasoning techniques such as induction are much more difficult to apply in the presence of infinite terms.

[2]for example, in Amzi!Prolog

11 Exercises

Exercise 1. Show precisely where and why the attempt to prove completeness of the rules for unification by mutual induction over the structure of t and $\mathbf{t}$ (instead of $t\sigma$ and $\mathbf{t}\sigma$) would fail.

Exercise 2. Show that the rules for unification terminate no matter whether given unifiable or non-unifiable terms t and s. Together with soundness, completeness, and determinacy of the rules this means that they constitute a decision procedure for finding a most general unifier if it exists.

References

[BS01] Franz Baader and Wayne Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 8, pages 447–532. Elsevier and MIT Press, 2001.

[Jaf84] Joxan Jaffar. Efficient unification over infinite terms. *New Generation Computing*, 2(3):207–219, 1984.

[Kni89] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[MM76] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[Pfe06] Frank Pfenning. Logic programming: Lecture 6: Unification. Lecture Notes 15-819K, Carnegie Mellon University, 2006.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sch12] Peter H. Schmitt. Formale Systeme. Vorlesungsskriptum Fakultät für Informatik , Universität Karlsruhe, 2012.