Lecture Notes on Unification

15-317: Constructive Logic
Frank Pfenning*

Lecture 18
November 3, 2016

In this lecture we investigate unification further to see that the algorithm from the previous lecture is also complete. These lecture notes are based on [7].

1 Completeness

Completeness of the 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 1 *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

*With edits by André Platzer

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

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

$(t_1, \mathbf{t}_2)\sigma = (s_1, \mathbf{s}_2)\sigma$	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 i.h.(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 **??**)
$\mathbf{t}_2\theta_1 \doteq \mathbf{s}_2\theta_1 \mid \theta_2$ and	
$\sigma'_1 = \theta_2\sigma'_2$ for some θ_2 and σ'_2	By i.h.(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 **??**)

□

It is worth observing that a proof by mutual induction on the structure of t and $\mathbf{t}$ would fail here (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 $\mathbf{s} = (s_1, \mathbf{s}_2)$, 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 $(\mathbf{t}_2\theta_1)\sigma'_1 = (\mathbf{s}_2\theta_1)\sigma'_1$.

2 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 non-determinism 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.

3 Historical Notes

Unification was originally developed by Robinson [8] 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 [1] 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 [4, 5] or dag structures with parent pointers [6]. These and similar algorithms are important in

some applications [3], 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[1]. 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 [2] 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.

4 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

[1] 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.

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

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

[1] for example, in Amzi!Prolog

[4] 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.

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

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

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

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