A Compositionality Principle for Syntactic Unification

Venkatesh Choppella
choppell@iiitmk.ac.in

Indian Institute of Information Technology and Management Kerala
and Tata Consultancy Services
Technopark, Thiruvananthapuram, Kerala 695 581, India

Abstract. Many unification based systems, including type reconstruction, logic programming, and theorem proving, operate by constructing and composing unifiers. Often, these unifiers are constructed in a non-compositional manner. This non-compositionality makes reasoning with these systems more cumbersome, specially during debugging. We propose an elementary compositionality principle for syntactic unification to avoid substitution-based non-compositional reasoning. The principle makes explicit the system of term equations to be unified and relates their union with unifier composition: if s_1 and s_2' are the most general unifiers of term equation sets E_1 and s_1E_2 respectively, then their composition $s_2's_1$ is the most general unifier of $E_1 \cup E_2$. This result allows a shift from a non-compositional computation of substitutions to a compositional construction of term equations. We have applied this principle to formalize the connection between substitution-based type reconstruction algorithms and equation-based algorithms and have designed a type-equation reformulation Milner's W algorithm for type reconstruction [1].

1 Introduction

Syntactic unification is at the heart of many symbolic computation systems, including automated deduction, term rewriting, logic programming, and type reconstruction.

In programming language semantics, much importance is placed on the property of *compositionality*. An evaluation function $[\![.]\!]$ is compositional if, for a compound expression $c(e_1, e_2)$ consisting of immediate subexpressions e_1 and e_2, the valuation of $c(e_1, e_2)$ is a function of the valuations of e_1 and e_2. In other words,

$$[\![c(e_1, e_2)]\!] = f([\![e_1]\!], [\![e_2]\!])$$

Compositionality requires that the valuation of e_2 *not* be dependent on the result of valuation of e_1 (and vice versa).

In many algorithms that rely on unification, the computation of unifiers violates compositionality. In this paper, we introduce a principle for syntactic unification that offers a way out of this non-compositionality. The principle relates

the union of term equations and composition of most general unifiers (mgu's): If s_1 and s_2' are the mgu's of term equation sets E_1 and $s_1 E_2$ respectively, then their composition $s_2' s_1$ is the mgu of $E_1 \cup E_2$. Using this principle, non-compositional computation of unifiers may be replaced with a compositional construction of term equations.

We have used the compositionality principle proposed here to prove the correctness of the W^E algorithm for type reconstruction. W^E is a reformulation of Milner's W polymorphic type reconstruction algorithm centered around the construction of type equations rather than substitutions and useful for source-tracking and diagnosing type errors [1].

2 Motivation: Source-tracking

Many unification based algorithms operate by constructing and composing substitutions. Examples include type reconstruction, logic programming, automated theorem proving, etc. Reasoning with substitutions, however, can be cumbersome. Often, the non-compositional construction of substitutions from intermediate substitutions introduces a sequentiality and asymmetry that makes formal reasoning awkward and difficult. Furthermore, since unifiers do not preserve the exact syntactic form of the equations they solve, it is difficult to directly use such algorithms for the purpose of *source-tracking*, tracing a program's execution in terms of its original source, specially in the case of errors. We consider two examples to illustrate our point: type reconstruction and logic programming.

2.1 Type reconstruction

Polymorphic type reconstruction in the Damas-Milner type system [2] is implemented using Milner's W algorithm [3]. W operates by computing substitutions (most general unifiers). As pointed out in McAdam [4], however, there is an apparent "left-to-right" bias introduced by W when handling the application construct. The code fragment below shows how the Milner algorithm W computes the type substitution for the application expression @ e_1 e_2 in the type environment A:

W : [TypeEnvironment, Expression] $\longrightarrow$ [TypeSubstitution, Type]
@ e_1 e_2:
let $\langle s_1, \tau_1 \rangle = W(A, e_1)$
  and $\langle s_2', \tau_2 \rangle = W(s_1 A, e_2)$
  and $u = mgu\{s_2'(\tau_1) \stackrel{?}{=} \tau_2 \rightarrow t\}$
    in $\langle u\ s_2'\ s_1,\ u(t) \rangle$
      where t is a new type variable

$W(@e_1 e_2, A)$ proceeds by computing the type substitution s_1 for e_1, and using s_1 to compute the type substitution (s_2') for e_2. A further substitution (mgu) u

is computed as the mgu of the equation $s'(\tau_2) \stackrel{?}{=} \tau_2 \to t$. The final substitution is computed as a composition $u\ s'_2\ s_1$ of the three intermediate substitutions. McAdam shows how the "left-to-right bias" resulting from this sequentiality of substitution computation results in difficult to understand type error messages returned by W.

The polymorphic type reconstruction algorithm W^E avoids the problem of non-compositionality by generating of type equations (equations over terms representing types) rather than substitutions. The type substitution is then obtained as the most general unifier of the system of type equations generated. W^E, like W takes an expression and a type environment, but returns a pointed set of type equations $\langle t, E\rangle$, where t is a type variable and E is a set of type equations. The details of the algorithm, its correctness and formal relation to W are in [1]. Here, we show only the code fragment of W^E implementing application:

W^E : [TypeEnvironment, Expression] $\longrightarrow$ [TypeVariable, TypeEquationSet]
@ $e_1\ e_2$:
let $\langle t_1, E_1\rangle = W^E(A, e_1)$
  and $\langle t_2, E_2\rangle = W^E(A, e_2)$
  and $E = E_1 \cup E_2 \cup \{t_1 \stackrel{?}{=} t_2 \to t\}$
    in $\langle t, E\rangle$
      where t is a new type variable

Note the compositional manner in which the resultant set of type equations E is computed from E_1 and E_2. The computation of E therefore avoids the sequentiality exhibited by W. Both W and W^E rely on unification. When the set of collected type equations E is unifiable, the mgu of the equations yields a type assignment. When E is non-unifiable, however, it is possible to analyze the source of non-unifiability of E, using techniques like unification source-tracking [5] that are independent of the mechanics of the type reconstruction algorithm. The construction and correctness of W^E relies on the compositionality principle presented in this paper.

2.2 Logic Programming

Standard implementations of logic programming languages like Prolog use an execution model that assumes a sequential "left-to-right" control structure. Let us, for example, consider the following the logic program:

$$\begin{array}{ll} \mathsf{f}(X, Z) & :- \mathsf{g}(X, Y), \mathsf{h}(Y, Z) \\ \mathsf{g}(\mathsf{a}, \mathsf{b}) & :- \\ \mathsf{h}(\mathsf{b}, \mathsf{c}) & :- \end{array}$$

A query is a term with logic variables. Solving a query returns a substitution (if it exists) over the logic variables in the term. In the Prolog execution model for the above logic program, solving the query $?\mathsf{f}(X, Z)$ returns the most general unifier $s_\mathsf{f} = \{X \mapsto \mathsf{a}, Z \mapsto \mathsf{c}\}$. The computation of s_f proceeds by first solving the

subgoal $\mathsf{g}(X, Y)$. This yields the most general unifier $s_\mathsf{g} = \{X \mapsto \mathsf{a}, Y \mapsto \mathsf{b}\}$. The computation then proceeds to solve the subgoal $s_\mathsf{g}\mathsf{h}(Y, Z)$ obtained by applying the intermediate unifier s_g to the term $\mathsf{h}(Y, Z)$, yielding the unifier $s_\mathsf{h} = \{Z \mapsto \mathsf{c}\}$. The final solution s_f is obtained as the composition $s_\mathsf{h} s_\mathsf{g}$. The application of the intermediate substitution s_g to $\mathsf{h}(Y, Z)$ makes the computation of s_f, the result of the original query $?\mathsf{f}(X, Z)$ non-compositional.

An alternate way to compute the query $?\mathsf{f}(X, Z)$ is to consider an evaluation model that builds systems of term equations instead of substitutions. Then, the evaluation of the subgoal $\mathsf{g}(X, Y)$ yields the *equation set* $E_\mathsf{g} = \{\mathsf{g}(X, Y) \stackrel{?}{=} \mathsf{g}(\mathsf{a}, \mathsf{b})\}$. The evaluation of the subgoal $\mathsf{h}(Y, Z)$ yields $E_\mathsf{h} = \{\mathsf{h}(Y, Z) \stackrel{?}{=} \mathsf{h}(\mathsf{b}, \mathsf{c})\}$. The final substitution s_f is computed as the most general unifier of the equation set $E_\mathsf{g} \cup E_\mathsf{h}$.

3 Related Research

Lassez et al. [6] examine syntactic unification from an algebraic point of view. Eder [7] studies algebraic properties of substitutions and unifiers. The unification algorithm of Martelli and Montanari [8] relies on transforming systems of equations to solved form and the proof of this algorithm rests on algebraic properties of unifiers. Surprisingly, none of these works allude to any compositionality principle for unification. To the best of our knowledge, the only work that addresses this problem is McAdam's paper on polymorphic type reconstruction algorithm W' based on the unification of substitutions [4]. In this approach, McAdam proposes a compositional solution in which the substitutions s_1 and s_2 computed from the calls $W'(A, e_1)$ and $W'(A, e_2)$ respectively are combined using a novel unification algorithm U_S. In contrast, our approach relies on a basic compositionality property of ordinary unification discussed in this paper. By working directly with systems of term equation sets, it also has the advantage of preserving source information (type equations in this case) from which the unifiers are derived.

4 Basic Definitions

The syntactic unification problem is concerned with solving equations between *terms* inductively defined over a denumerable set of *variables* V and a fixed signature Σ consisting of a set of constructor symbols and an *arity* function mapping each constructor to a natural number. A *substitution* s is a finite mapping from V to terms over V. The domain of a substitution s is indicated $dom(s)$. The extension $\hat{s}$ of s is a function from terms to terms defined inductively in the following way: if τ is a term equal to a variable $x \in V$, then $\hat{s}(\tau)$ is equal to $s(x)$ if $x \in dom(s)$, and x if $x \notin dom(s)$. Otherwise, if τ is the term $f(\tau_1, \ldots, \tau_n)$, where f is a constructor symbol, n its arity, and $\tau_1, \ldots, \tau_n$ are terms, then $\hat{s}(\tau) = f(\hat{s}(\tau_1), \ldots, \hat{s}(\tau_n))$. We will often overload s to denote its extension $\hat{s}$. Substitutions s_1 and s_2 are considered equal if $\hat{s_1}(x) = \hat{s_2}(x)$ for each $x \in V$.

If s, s' are two subsitutions, then their *composition* $s's$ is the substitution with domain $dom(s) \cup dom(s')$ defined as $s's(x) = \hat{s}(\hat{s}(x))$, for each $x \in dom(s's)$. A substitution is *idempotent* if $s = ss$. Idempotent substitutions are not closed over composition.

If $E = \{\tau_i \stackrel{?}{=} \tau_i'\}$ is a set of term equations, a subsitution s is a *unifier of* E if for each $\tau_i \stackrel{?}{=} \tau_i' \in E$, $s(\tau_i) = s(\tau_i')$. s is a *most general unifier (mgu) of* E if for any unifier s' of E, there is a substitution α such that $s' = \alpha s$. s is an *idempotent most general unifier (imgu) of* E if s is an mgu of E and is idempotent. Given a set of equations E and a substitution s, sE denotes the set of equations $\{s(\tau) \stackrel{?}{=} s(\tau') \mid \tau \stackrel{?}{=} \tau' \in E\}$.

5 The Compositionality Principle

We start by recounting the following relation between term equation sets, their unifiers, and substitutions. The first part of Lemma 1 states that unifiers are closed under composition with substitutions. The second part shows how to obtain a unifier of a set of term equations E from a substitution s and a unifier s' of sE.

Lemma 1. *Let E denote a set of term equations, and let s,s' denote substitutions.*

1. *If s is a unifier of E, and s' is a substitution, then $s's$ is a unifier of E.*
2. *If s is a substitution and s' is a unifier of sE, then $s's$ is a unifier of E.*

Proof.
- (1): If s is a unifier of E, then $s\tau = s\tau'$ for each $\tau \stackrel{?}{=} \tau' \in E$. Hence if s' is a substitution then $s's\tau = s's\tau'$, implying $s's$ is a unifier of E.
- (2) Again $s's\tau = s's\tau'$ for each $\tau \stackrel{?}{=} \tau' \in E$. Hence $s's$ is a unifier of E.

Theorem 1 (Compositionality Principle for Syntactic Unification). *Let E_1, E_2 be sets of term equations, and s_1, s_2' be substitutions. If s_1 is a unifier (mgu, imgu) of E_1, and s_2' is a unifier (respectively mgu, imgu) of s_1E_2, then $s_2's_1$ is a unifier (respectively mgu, imgu) of $E_1 \cup E_2$.*

Proof. Due to (1), $s_2's_1$ is a unifier of E_1, and due to (2), $s_2's_1$ is a unifier of E_2. Hence, $s_2's_1$ is a unifier of $E_1 \cup E_2$. Now suppose that s_1 is an mgu of E_1 and s_2' is an mgu of s_1E_2. We show that $s_2's_1$ is an mgu of $E_1 \cup E_2$ by showing that if s is any mgu of $E_1 \cup E_2$, then $s_2's_1$ is more general than s. Since s unifies $E_1 \cup E_2$, it unifies E_1 and also E_2. Since s_1 is an mgu of E_1, it follows that s_1 is more general than s, that is, $s = rs_1$ for some substitution r. Since $s = rs_1$, this means that rs_1 unifies E_2. That is, $rs_1(\tau) = rs_1(\tau')$ for each equation $\tau \stackrel{?}{=} \tau' \in E_2$. Hence r is a unifier of s_1E_2. Since s_2' is an mgu of s_1E_2, it follows that s_2' is more general than r, and therefore $r = ps_2'$ for some substitution p. Hence, we have $s = rs_1 = ps_2's_1$, which implies that $s_2's_1$ is more general than s. This proves that $s_2's_1$ is an mgu of $E_1 \cup E_2$.

Now assume that s_1 and s_2' are both idempotent mgu's. We show that $s_2's_1$ is idempotent. Since s_2' is a unifier of s_1E_2, it follows that $s_2's_1(\tau) = s_2's_1(\tau')$, for each $\tau \stackrel{?}{=} \tau' \in E_2$. Since s_1 is idempotent, this means that $s_2's_1s_1(\tau) = s_2's_1s_1(\tau')$, and hence $s_2's_1$ is a unifier of s_1E_2. Since s_2' is an idempotent mgu of s_1E_2, it follows that $s_2's_1 = s_2's_1s_2'$. We use this equation to simplify the expression $s_2's_1s_2's_1$ to $s_2's_1s_1$, which further simplifies to $s_2's_1$, since s_1 is idempotent. Thus, $s_2's_1s_2's_1 = s_2's_1$, implying $s_2's_1$ is idempotent.

6 Conclusions and Future Work

We have stated and proved a compositionality principle for syntactic unification that allows exchanging unifier composition with term equation union. The utility of this principle is illustrated with a brief reference to two examples: a reformulation of the polymorphic type reconstruction algorithm of Milner, and evaluation in Prolog.

We expect this principle to be used in other domains that rely on unifier composition, like automated deduction and intelligent backtracking. We are currently exploring the role of this principle in intelligent backtracking in logic programming languages.

References

1. V. Choppella, Polymorphic type reconstruction using type equations, in: G. Michaelson, P. Trinder (Eds.), Selected Papers from the 15th International Workshop on the Implementation of Functional Languages (IFL 2003), LNCS, Springer, 2004, (To appear).
2. L. Damas, R. Milner, Principal type-schemes for functional languages, in: Proc. 9th ACM Symp. on Principles of Programming Languages, 1982, pp. 207–212.
3. R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17 (1978) 348–375.
4. B. J. McAdam, On the unification of substitutions in type inference, in: K. Hammond, A. J. T. Davie, C. Clack (Eds.), International Workshop on the Implementation of Functional Languages, 1998, Vol. 1595 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 139–154.
5. V. Choppella, C. T. Haynes, Source-tracking Unification, in: F. Baader (Ed.), Proceedings of 19th International Conference on Automated Deduction, CADE-19, Miami Beach, USA, no. 2741 in Lecture Notes in Artificial Intelligence, Springer, 2003, pp. 458–472.
6. J. Lassez, M. J. Maher, K. Marriot, Unification revisited, in: J. Minker (Ed.), Deductive Databases and Logic Programming, Morgan Kaufmann, 1988, Ch. 15, pp. 587–625.
7. E. Eder, Properties of substitutions and unifiers, Journal of Symbolic Computation 1 (1985) 31–46.
8. A. Martelli, U. Montanari, An efficient unification algorithm, ACM Trans. Prog. Lang. Syst. 4 (2) (1982) 258–282.