Unification Modulo Common List Functions

(Extended Abstract)

Peter Hibbs*, Paliath Narendran* and Shweta Mehto

Department of Computer Science
University at Albany–SUNY
Albany, NY 12222

1 Introduction

Reasoning about data types such as lists and arrays is an important research area with many applications, such as formal program verification [12, 10]. Early work on this [7] focused on proving inductive properties. Important outcomes of this work include the technique of *proof by consistency* or "inductionless induction" [13, 11] as well as *satisfiability modulo theories* (SMT) starting with the pioneering work of Nelson and Oppen [14] and of Shostak [16]. (See [1] for a recent syntactic, inference-rule based approach to developing SMT algorithms for lists and arrays.)

In our paper, we investigate the *unification* problem modulo various theories of lists. The constructors used in this paper are the usual nil and cons. The different theories are obtained by considering observer functions of increasing complexity. We first examine lists with only *right cons* (*rcons*) as an operator and propose a novel algorithm for the unification problem for this theory. We then introduce the theory of *reverse* (*rev*) and develop an algorithm to solve the unification problem over this theory. Lastly, we consider the unification problem modulo the theory of *fold right* or *reduce* which is of central importance in functional programming languages [9]. Note that in practice *reduce* is not a first-order function; we turn it into a first-order function by treating the binary function to be "folded" over the list as an uninterpreted function, i.e., as a constructor.

2 Definitions

The reader is assumed to be familiar with the concepts and notation used in [2]. For terminology and a more in-depth treatment of unification the reader is referred to [4]. Due to space constraints, we omit many proofs and details. Interested readers are referred to our technical report [8] in which much more detail is given.

The unification problems we consider are instances of *unification with constants* with some caveats. For instance, we only consider nil-terminated lists — this means that for any ground term X, the innermost element of X must be nil[1]. Constants of the list type are not permitted.

3 *rcons*

The equational axioms of this theory are

*supported in part by NSF grant CNS 09-05286.

[1] In LISP parlance, these are called *proper lists*.

$$\begin{aligned} rcons(\mathsf{nil}, x) &= \mathsf{cons}(x, \mathsf{nil}) \\ rcons(\mathsf{cons}(x, y), z) &= \mathsf{cons}(x, rcons(y, z)) \end{aligned}$$

We refer to this equational theory as RCONS and orient these from left to right to produce a convergent rewrite system.

In addition to requiring nil-terminated lists, we enforce the further restriction that the lists be *homogeneous* as in ML. We consider a *typed* theory here with a base sort of non-list elements. The lists we consider are of type list and may contain either list or non-list elements. Lists which do not themselves contain lists are called *flat lists*.

3.1 Complexity Analysis

Lemma 3.1. *Let s_1, s_2, t_1, t_2 be terms such that*

$$rcons(s_1, t_1) =_{\text{RCONS}} rcons(s_2, t_2).$$

Then $s_1 =_{\text{RCONS}} s_2$ and $t_1 =_{\text{RCONS}} t_2$.

Theorem 3.2. *Unifiability modulo* RCONS *is NP-hard.*

Proof Sketch. We will show this by reduction from monotone 1-in-3-SAT using the following device:

$$S_i : \ \mathsf{cons}(0, \mathsf{cons}(0, \mathsf{cons}(1, L_i))) =^? rcons(rcons(rcons(L_i, x_i), y_i), z_i)$$

Note that the solution to S_i must have exactly one of $\{x_i, y_i, z_i\}$ mapped to 1 and the others to 0. □

To show membership in NP we first consider the case where the lists are *flat*. Thus we only have two kinds of variables: variables of type list and variables of type non-list. We guess equivalence classes of all of the variables of type non-list. We consider these equivalence classes to be *discriminating*. That is, we fail on equations of the form $X =^? Y$ where X, Y are from different equivalence classes. All variables of type non-list may therefore be treated as constants.

Once this step is done, all equations are expressible in the following way:

$$\begin{aligned} &\mathsf{cons}(a_1, ...(\mathsf{cons}(a_n, rcons(rcons(...rcons(X, b_n)..., b_1))))) =^? \\ &\mathsf{cons}(c_1, ...(\mathsf{cons}(c_n, rcons(rcons(...rcons(Y, d_n), ..., d_1))))) \end{aligned}$$

with X and Y not necessarily distinct. We will denote the sequences $\{a_i\}$, $\{b_i\}$, $\{c_i\}$, $\{d_i\}$ with α, β, γ, δ respectively. We thus have to solve equations of the form $\alpha X \beta \ =^? \ \gamma Y \delta$.

Lemma 3.3. *The following algorithm can be used to solve the problem. Unification modulo* RCONS *is therefore NP-Complete*

1. For each equation in U, $\alpha X \beta =^? \gamma Y \delta$, remove all common prefixes and suffixes from both sides of that equation.

2. Select an arbitrary equation such that the variables on the right and left hand sides of the equation are distinct. If no such equation is available, proceed to Step 5.

3.a. If the equation is of the form $X =^? \alpha Y \beta$, replace all instances of X by $\alpha Y \beta$.

3.b. If the equation is of the form $\alpha X =^? Y\beta$, there is always a solution to X, Y of the form $X \mapsto Z\beta, Y \mapsto \alpha Z$, where Z is a fresh variable. If there exists some set of strings $\{u, v, w\}$ where $\alpha = uv$ and $\beta = vw$ where $v \neq \epsilon$ then there is an additional solution: $\{X \mapsto u, Y \mapsto w\}$. This solution is checked for validity. If there is more than one such set of strings, they are all checked. If no valid solution is found, replace all instances of X and Y with $Z\beta$ and αZ respectively. The number of variables is thereby reduced by 1.

4. Repeat from Step 1.

5. We are now left with only equations in solved form, and *independent* systems of equations each of which has only *one variable* in it. These can be checked for solvability using the algorithm in [5].

We omit the extension to *non-homogeneous, non-flat lists* here and direct the reader to [8].

4 *rev*

The term rewriting system we consider for nil-terminated lists is

$$\begin{array}{lrcl}
(1) & rcons(\mathsf{nil}, x) & \to & \mathsf{cons}(x, \mathsf{nil}) \\
(2) & rcons(\mathsf{cons}(x, y), z) & \to & \mathsf{cons}(x, rcons(y, z)) \\
(3) & rev(\mathsf{nil}) & \to & \mathsf{nil} \\
(4) & rev(\mathsf{cons}(x, y)) & \to & rcons(rev(y), x) \\
(5) & rev(rcons(x, y)) & \to & \mathsf{cons}(y, rev(x)) \\
(6) & rev(rev(x)) & \to & x
\end{array}$$

This system is convergent. We refer to this equational theory as REV. From this point on, we consider all terms to be in normal form modulo this term rewrite system.

Lemma 4.1. *Let s_1, s_2, t_1, t_2 be terms such that $rcons(s_1, t_1) =_{\text{REV}} rcons(s_2, t_2)$. Then $s_1 =_{\text{REV}} s_2$ and $t_1 =_{\text{REV}} t_2$.*

Lemma 4.2. *Let s_1, s_2 be terms such that $rev(s_1) =_{\text{REV}} rev(s_2)$. Then $s_1 =_{\text{REV}} s_2$.*

Lemma 4.3. *Unifiability modulo* REV *is NP-Complete.*

The NP-hardness proof given for unifiability modulo RCONS is equally valid for unifiability modulo REV. Membership in NP is shown by providing an algorithm to solve unification modulo REV which runs in NP time: we first guess equivalence classes of our variables as in the previous section. We then remove the 'highest' applications of *rev* in the dependency graph by applying the following inference rules:

(r1) $$\frac{\mathcal{EQ} \uplus \{X =^? rev(Y),\ X =^? \mathsf{cons}(W, Z)\}}{\mathcal{EQ} \cup \{X =^? \mathsf{cons}(W, Z),\ Y =^? rcons(Z', W),\ Z =^? rev(Z')\}}$$

(r2) $$\frac{\mathcal{EQ} \uplus \{X =^? rev(X),\ X =^? \mathsf{cons}(Y, Z)\}}{\mathcal{EQ} \cup \{X =^? \mathsf{cons}(Y, Z),\ Z =^? rcons(Z', Y),\ Z' =^? rev(Z')\}} \quad \text{if } Z \neq \mathsf{nil}$$

(r3) $$\frac{\mathcal{EQ} \uplus \{X =^? rev(X),\ X =^? \mathsf{cons}(Y, \mathsf{nil})\}}{\mathcal{EQ} \cup \{X =^? \mathsf{cons}(Y, \mathsf{nil})\}}$$

Each of the above rules (r1-r3) have analogous *rcons* rules which are very similar to the ones given here and are therefore omitted. After the above rules are applied to termination, the applications of rev exist only on the variables which correspond to leaf-nodes in the dependency graph. We now apply the rules of the flat case but once we have removed all equations of the form $\alpha X\beta =^? \alpha' Y\beta'$ where $X \neq Y$ and $Y \neq X^R$ where X^R denotes $rev(X)$, we then move on to *palindrome discovery*. In this step, we consider all equations of the form $\alpha X\beta =^? \alpha' X^R\beta'$. We maintain a list of variables that are known to be palindromes (i.e., where $X = X^R$) which is initially empty. We now have two cases:

> Case 1: $X =^? \alpha'' X^R\beta''$ in this case, if $|\alpha''\beta''| = 0$, then we conclude that X is a palindrome. Else, there can be no solution and we terminate with failure.
>
> Case 2: $\alpha'' X =^? X^R\beta''$. In this case, we check for the existence of a pair of strings $\{u, v\}$ such that $\alpha'' = u^R v, \beta = vu$. If such a pair exists, we check $X = u$ for consistency. If all such pairs are checked without finding a solution, then we default to the substitution $X = Z\beta''$, $X^R = \alpha'' Z$ where Z is known to be a palindrome. If $\beta'' \neq \alpha''^R$, then there is no solution and we fail. Otherwise we replace all occurrences of X with $Z\beta''$.

Once we have finished this, we only have equations of the form $\alpha X =^? X\beta$. If X is not a palindrome, then we may use the algorithm given in [5] to find a solution for it. If X is known to be a palindrome, then we may still run the algorithm given in [5] to check for a solution, but first check that the prefixes and suffixes of each equation (i.e., α, β) meet certain criteria:

Lemma 4.4. *Let α, β and A be non-empty strings such that A is a palindrome and $|\alpha| = |\beta| < |A|$. Then $\alpha A = A\beta$ if and only if there exist palindromes u, v, and a positive integer k such that $\alpha = uv$, $\beta = vu$ and $A = (uv)^k u$.*

Proof Sketch. This follows from the well-known result that for any equation $\alpha A = A\beta$ where $0 < |\alpha| = |\beta| < |A|$, α and β must be *conjugates* or there can be no solution. □

So, if the elements in the set of equations satisfy this constraint, then any solution must be a palindrome. Thus it is sufficient to check for the existence of appropriate u, v and then apply the algorithm of [5].

Lemma 4.5. *The above algorithm terminates*

Proof Sketch. The algorithm begins by applying inference rules (r1-r3) to termination. Each of these rules either lowers some application of rev further down in the dependency graph or deletes it outright. Because the set of input equations is finite, eventually all applications of rev must lie on the leaf-nodes of the graph and no further lowering can occur. The algorithm then removes all equations of the form $\alpha X\beta =^? \alpha' Y\beta'$ where $X \neq Y$ and $Y \neq X^R$ which terminates by the argument given in the statement of this procedure in Section 3.1. We then move on to the palindrome discovery step which removes an equation of the form $\alpha X\beta =^? \alpha' X^R\beta'$ in each iteration. Finally, we apply the algorithm given in [5] which terminates by assumption. □

5 *reduce*

The standard definition of *reduce* (for a particular two-argument function f) is given by the following rewrite rules:

$$\begin{aligned} reduce(\mathsf{nil}, x) &\rightarrow x \\ reduce(\mathsf{cons}(u, v), x) &\rightarrow f(u,\, reduce(v, x)) \end{aligned}$$

Since we consider only nil-terminated lists, we extend the signature of the theory with the *append* function @ and a monadic function g which creates *singleton* lists. This extended theory has the following convergent rewrite system:

$$\begin{array}{lrcl}
(1) & f(x,z) & \to & reduce(\mathsf{g}(x),z)\\
(2) & \mathsf{cons}(x,y) & \to & \mathsf{g}(x)\ @\ y\\
(3) & reduce(\mathsf{nil},z) & \to & z\\
(4) & reduce(x,reduce(y,z)) & \to & reduce(x\ @\ y,z)\\
(5) & \mathsf{nil}\ @\ x & \to & x\\
(6) & x\ @\ \mathsf{nil} & \to & x\\
(7) & (x\ @\ y)\ @\ z & \to & x\ @\ (y\ @\ z)
\end{array}$$

Note that $\mathsf{g}(x)$ is equivalent to $\mathsf{cons}(x, \mathsf{nil})$. We impose a type system for this equational theory. There are two types: `list` and `nonlist`. Under this type system the unification problem $\{reduce(X,Y) =^? \mathsf{cons}(U,V)\}$, for example, will result in a type-failure. Unification modulo this theory is at least as hard as the word equation problem, which is NP-hard and in PSPACE [15].

We now outline the algorithm to solve the unification problem modulo the extended theory. We assume the input equations are in standard form. We also assume that all instances of the function symbols f and cons are eliminated using the rewrite rules (1) and (2).

Let S be the set of `list` type variables. As in Section 3.1, we nondeterministically guess a partition of equivalence classes among all variables. We guess an ordering $\succ$ on the `list` type equivalence classes such that $X \succ Y$ if the length of X is larger than the length of Y, where the length of a variable Z refers to the number of instances of cons in Z. All list variables in the same equivalence class as nil must be equivalent to nil and clearly the partition containing nil must be a least element in the ordering $\succ$. We also nondeterministically guess an ordering $\gg$ on the `nonlist` variables, just as with the `list` variables, such that $X \gg Y$ if and only if the size of X after substitution is greater than the size of Y. This ordering is clearly acyclic and well-founded.

Lemma 5.1. *If $X = reduce(Y,Z)$ and Y is not equivalent to* nil *then $X \gg Z$.*

Proof Sketch. We prove this by induction on the length of the `list` variable Y. □

From this point on, if at any time in the algorithm an equation violates a type constraint or an ordering constraint, we terminate with failure. The inference rules for those failures are not included. We apply rewrite rules (3), (5) and (6) to remove equations involving nil. After this, once nils are eliminated, the problem boils down to unification modulo the rules

$$\begin{array}{lrcl}
(4) & reduce(x,reduce(y,z)) & \to & reduce(x\ @\ y,z)\\
(7) & (x\ @\ y)\ @\ z & \to & x\ @\ (y\ @\ z)
\end{array}$$

No rule has nil on the right-hand side (thus new instances of nil will not be produced) and, since g does not occur in these rewrite rules, the problem is now a *general* unification problem.

We construct a dependency graph for our unification problem U. If this graph contains a cycle, then clearly U is not unifiable unless the above theory is subterm-collapsing which it is not. Thus, if there is a cycle we terminate with failure. The main inference rule is

$$\textbf{(5)}\qquad \frac{\mathcal{EQ} \uplus \{X =^? reduce(Y,Z),\ X =^? reduce(V,W)\}}{\mathcal{EQ} \cup \{X =^? reduce(Y,Z),\ Y =^? V@Y',\ W = reduce(Y',Z)\}} \quad \text{if } Y \succ V$$

Note that we introduce a (possibly) new `list`-type variable Y' in rule (5). At that point we nondeterministically include Y' into the ordering $\succ$. (We omit failure rules here.) The only equations now left that are not in solved form are equations of the form $X =^? Y@Z$. Thus the set of equations we get is an instance of the *general* associative unification problem, which is decidable [3].

The termination and correctness of this algorithm is given in the technical report [8].

6 Conclusions

We have shown that unification of lists modulo the observer functions *rcons* and *rev* is NP-complete. Our algorithm for unification modulo *reduce* requires solving the general associative unification problem, and the algorithm for the latter makes use of an algorithm for the word equation problem with rational (regular) constraints. This problem (i.e., word equations with rational constraints) has been shown to be solvable in PSPACE [6] and our algorithm therefore requires no more than PSPACE complexity. The lower bound on the complexity of this last problem is an open question.

Acknowledgements: We wish to thank Dan DiTursi, Kim Gero, Wojciech Plandowski, Manfred Schmidt-Schauß and the referees for their helpful comments and suggestions.

References

[1] A. Armando, S. Ranise, M. Rusinowitch. Uniform Derivation of Decision Procedures by Superposition. *Lecture Notes in Computer Science* 2142: 513–527 (2001)

[2] F. Baader, T. Nipkow. *Term Rewriting and All That.* Cambridge Univ Press, 1999.

[3] F. Baader and K.U. Schultz. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *Proceedings of the Eleventh Conference on Automated Deduction (CADE-11),* Saratoga Springs, New York, *Lecture Notes in Artificial Intelligence* **607** (Springer, Berlin, 1992) 50–65.

[4] F. Baader, W. Snyder. Unification Theory. In: John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[5] R. Dabrowski, W. Plandowski. On Word Equations in One Variable. *Algorithmica* 60(4): 819–828 (2011).

[6] V. Diekert, A. Je, W. Plandowski. Finding All Solutions of Equations in Free Groups and Monoids with Involution. *arXiv* :1405.5133 [cs.LO]

[7] J.V. Guttag, E. Horowitz, D.R. Musser. Abstract Data Types and Software Validation. *Commun. ACM* 21(12): 1048–1064 (1978).

[8] P. Hibbs, P. Narendran, S. Mehto. Unification Modulo Common List Functions. Technical Report SUNYA-CS-14-01, available at: `www.cs.albany.edu/~ncstrl/treports/Data/`

[9] G. Hutton. A tutorial on the universality and expressiveness of *fold*. *Journal of Functional Programming* 9(4): 355–372 (1999)

[10] D. Kapur. *Towards A Theory For Abstract Data Types.* Doctoral Dissertation, Massachusetts Institute of Technology, 1980.

[11] D. Kapur, D.R. Musser. Proof by Consistency. *Artif. Intell.* 31(2): 125–157 (1987).

[12] D.R. Musser. Abstract Data Type Specification in the AFFIRM System. *IEEE Trans. Software Eng.* 6(1): 24–32 (1980).

[13] D.R. Musser. On Proving Inductive Properties of Abstract Data Types. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages* (POPL) 154–162 (1980).

[14] G. Nelson, D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems* 1(2): 245–257 (1979).

[15] W. Plandowski. An Efficient Algorithm For Solving Word Equations. In: *Proceedings of the ACM Symposium on the Theory of Computing '06*, 467–476 (2006).

[16] R.E. Shostak. Deciding Combinations of Theories. *J. ACM* 31(1): 1–12 (1984)